String Allocation in Icon

Ralph E. Griswold

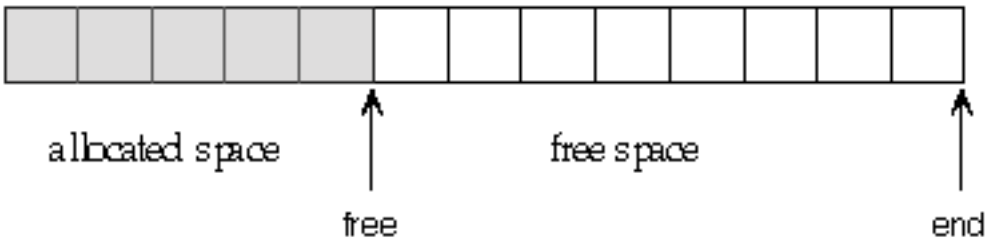
Department of Computer Science
The University of Arizona
Tucson, Arizona

IPD277
May 12, 1996
http://www.cs.arizona.edu/icon/docs/ipd275.html

Note: This report is an adaptation of an article that appeared in Issue 9 of the *Icon Analyst* in 1991.

Background

Strings that are created during the execution of an Icon program are stored at a place in memory called the string region. Strings in the string region are allocated contiguously, starting at the beginning. Thus, the string region is divided into two portions: an allocated portion and a free portion. Whenever a new string is created, it is added to the end of the allocated portion of the string region, decreasing the size of the free portion. The string region can be depicted as a long sequence of characters:



where free identifies the boundary between the allocated space and the free space. Of course, there are many more characters in the string region than suggested by this diagram -- typically 65K of them.

If the free portion is not large enough to hold a newly created string, a garbage collection is performed, discarding strings that are no longer needed and compressing the allocated part, making more room in the free part. See Reference 1 for more information about the details of allocation and garbage collection.

Strings created during program execution, called allocated strings, are not null-terminated as they are in C and some other programming languages. Null-termination, which adds a character with code zero to the end of a string, is used as a means of finding the end of a string. Icon accomplishes this by keeping both a pointer to the first character of a string and its length in an Icon string value.

This method makes computation of the length of a string fast, and it also allows null characters to appear in Icon strings. Perhaps more important, a substring of an existing string can be formed by changing only the pointer to the first character and the length, whereas with null termination, it's generally necessary to copy a portion of the existing string, since a terminating null character would overwrite a character in the string of which it's a part. A substring operation in Icon such as

```
s[i:j]
```

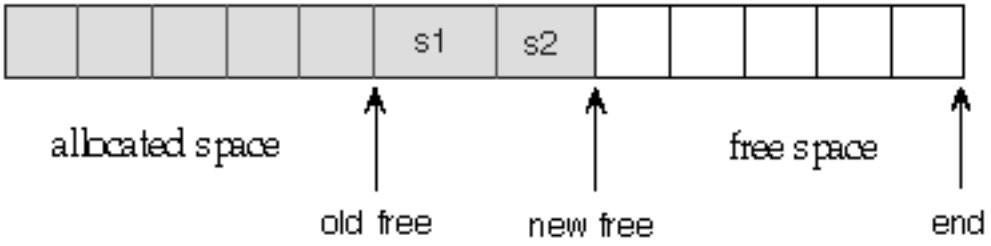
does not allocate any space in the string region. Several Icon operations do allocate space in the string region, the most notable being reading and concatenation.

Concatenation

Concatenation works by copying its two argument strings to the end of the allocated portion of the string region. The result of

```
s1 || s2
```

can be depicted as follows:



In general, such a concatenation allocates space for

```
*s1 + *s2
```

characters and copies that many characters. One of the problems with concatenating in this fashion is illustrated by the following program segment that builds up a string piece by piece:

```
result := ""
while s := read() do
   result := result || s
```

In the loop, space is allocated for the string that is read, which then is assigned to `s`. The concatenation then allocates space for result and s, and the resulting string is assigned back to result. Note that all new string data comes from reading. If the successive strings read are indicated by subscripts, the pattern of allocation is

```
s1 s1 s2 s1 s2 s3 s1 s2 s3 s4 s1 s2 s3 s4 ...
```

Every previously read string is copied to perform the next concatenation. Although the previous values of result are discarded when a garbage collection occurs, the amount of space allocated is clearly much larger than is needed for the final result. In addition, garbage collection can be expensive, and the cost of garbage collection may depend on other things that have gone on during program execution, not just on the concatenation loop.

If you look at the final result of the concatenation loop, you'll see it's just

```
s1 s2 s3 s4 ...
```

The rest is "garbage". Is all the intermediate allocation necessary? That's the kind of question that the implementation of Icon attempts to address.

Optimizations

Short of changing the way concatenation, and hence string allocation, is done, the question becomes a more general one: "Are there situations in which it's not necessary to allocate all of the space needed in the most general case?"

One situation is when one or both of the arguments of concatenation is the empty string, in which case no concatenation or allocation is necessary. We'll come back to this case later.

There are two other situations that lend themselves to optimizations.

Optimization 1: The code segment discussed above illustrates a situation in which an optimization is possible. As the loop is evaluated repeatedly, result is the last string allocated when another string is read in and assigned to s by the next iteration of the loop. After reading, but before concatenating, the last two allocated strings are result and s. But that's exactly what's needed for the concatenation!

In other words, because of the order in which strings are allocated, result and s are already concatenated. And it doesn't take much to check for this situation, since an Icon string value contains a pointer to its first character and the length. For the concatenation

```
s1 || s2
```

pseudo-code for the check looks like this:

```
if loc(s1) + len(s1) = loc(s2) then ... # done
```

If this test succeeds, no allocation is done, the new value points to `s1`, and its length is the sum of the lengths of `s1` and `s2`. With this optimization, the pattern of allocation for the loop given earlier is

```
s1 s2 s3 s4 ...
```

which is exactly what's needed for the final result, with no extra allocation. Note that the test above is more general, and applies anywhere in the allocated portion of the string region, although the chance of its succeeding anywhere but for the last two allocated strings is small.

Optimization 2: There's another situation in which part of the allocation for concatenation can be avoided -- when the first argument of the concatenation is the last allocated string and hence at the end of the allocated portion of the string region:

```
if loc(s1) + len(s1) = free then ... # don't copy
```

In this case, it's only necessary to append the second argument of the concatenation to the end of the string region. A situation in which Optimization 2 applies is shown by

```
result := ""
while s := read() do
   result := s || result
```

In the absence of Optimization 2, the pattern of allocation for the loop is

```
s1 s1 s2 s2 s1 s3 s3 s2 s1 s4 s4 s3 s2 s1 ...
```

With Optimization 2, the allocation pattern is

```
s1 s2 s1 s3 s2 s1 s4 s3 s2 s1 ...
```

The savings aren't as great as for Optimization 1; you can't expect to reverse the order of strings, which is what is happening here, without some copying.

Optimization 2 has a long standing. It was first used, to our knowledge, in the SPITBOL implementation of SNOBOL4 [4] and was incorporated in the first implementation of Icon.

Other Optimizations

Earlier we skipped over the situation in which one or both of the arguments of concatenation is the empty string. This sounds like a situation worth checking, since when it occurs, no allocation is necessary.

There's a kicker, however: The test has to be applied for every concatenation. It turns out that the cumulative cost of checking for this situation takes more time on the average than it saves (which is not true for Optimizations 1 and 2).

There's a moral here: An optimization may sound good, but the cost of testing for a special case may outweigh the savings when it does apply.

Some interesting examples of this occur in an implementation of SNOBOL4 that included some clever "heuristics" that turned out to be unfortunate in practice [2].

The trouble is that optimizations usually can't be evaluated analytically. And it may be very time-consuming and expensive to evaluate them in practice. The natural tendency is to rely on an intuitive feeling of the usefulness of an optimization. Such intuition often is faulty.

Other Allocation Strategies

If you think about how Icon allocates string space, you'll notice that the same string may occur in many different places in Icon's allocated string region.

Occasionally someone suggests that before allocating a new string, there should be a search to see if it already exists. This certainly is a bad idea -- it's very expensive to perform character comparisons just on the chance of finding a copy of a string that can be "re-used".

A different idea would be to put all strings in a hash table [3], so that each different string would be allocated only once. Although there are some fast and clever hashing techniques, they eventually come down to character comparison, which is quite expensive compared to an occasional garbage collection to remove unused strings. Furthermore, in a hashing scheme, the sharing of characters among substrings is not possible.

But again, the only way to be sure about this is to actually try it (or possibly simulate it, although a simulation in a case like this is difficult and error-prone). This would be a *major* effort, and one hardly worth undertaking, especially when the expectation clearly is negative.
.

Taking Advantage of the Optimizations

In most cases, the optimizations for string concatenation, like other aspects of the implementation of Icon, are "just there" and you needn't think about them when you program.

On the other hand, if you like to hone your programs for maximum performance, you may want to give a little thought to taking advantage of the concatenation optimizations.

One thing to watch out for is intermediate allocation that defeats the optimizations. For example, in

```
result := ""
while s := read() do {
      write(repl("=", *s))
      result := result || s
      }
```

the allocation for

```
repl("=", *s)
```

follows the allocation for

```
read()
```

and defeats Optimization 1 that otherwise would apply. On the other hand, other strings can be appended to the concatenation without additional allocation, as in

```
result := ""
while s := read() do
      result := result || s || ","
```

Here, Optimization 1 applies to the first (left) concatenation, while Optimzation 2 applies to the second (right) concatenation. It's worth knowing that literal strings are contained in the code produced by compiling a program, and space for using them is not allocated in the string region unless they have to be copied in concatenation or some other operation that allocates space in the string region [5]. For example,

```
marker := "+"
```

does not allocate space, but

```
marker := marker || "+"
```

does. One situation in which Optimization 1 may apply at a place other than at the end of the string region occurs in string scanning. For example, in

```
text ? {
   if t := tab(many(&letters)) then
      t := t || tab(upto(' ') | 0)
   }
```

`tab(many(&letters))` and `tab(upto(' ') | 0)` are adjacent in `text`, which need not be at the end of the allocated portion of the string region.

Output as a Weak Form of Concatenation

Jim Gimpel likes to call output a "weak form of concatenation". His point is that when a string is written to a file sequentially, it is automatically appended to (concatenated onto) the last string written.

As shown above, the concatenation of strings in a program requires the allocation of space (which may eventually result in a garbage collection) and also the copying of characters. This is costly compared to, say, arithmetic.

In many programs, most strings that are built up by concatenation are eventually written to a file. If you can arrange to write the strings as they are computed and in the order they need to be output, you can save the costs involved in concatenation.

As a very simple example, it's considerably more efficient to use

```
write(s1, s2)
```

than to use

```
write(s1 || s2)
```

It's often possible to avoid actual concatenation altogether in programs that transform input data. Sometimes all it takes to use output instead of concatenation is looking at the problem in the right way. In fact, it can be fun to see how far you can carry this.

References

1. "Memory Monitoring", *The Icon Analyst 2* , October 1990, pp. 5-9.

2. "Performance of Storage Management in an Implementation of SNOBOL4", David G. Ripley, Ralph E. Griswold, and David R. Hanson, *IEEE Transactions on Software Engineering* , Vol. SE-4, No. 2 (1978), pp. 130-137.

3. The *Macro Implementation of SNOBOL4; A Case Study of Machine-Independent Software Development*, Ralph E. Griswold, W. H. Freeman, San Francisco, California, 1972.

4. "MACRO SPITBOL -- A SNOBOL4 Compiler", Robert B. K. Dewar and Anthony P. McCann, *Software -- Practice & Experience* , Vol. 7 (1977), pp. 95-113.

5. "An Imaginary Icon Computer", *The Icon Analyst 8* , October 1991, pp. 2-6.

Icon home page