SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(12), 1315–1330 (DECEMBER 1995)

Ropes: an Alternative to Strings

HANS-J. BOEHM, RUSS ATKINSON AND MICHAEL PLASS
Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304, U.S.A. (email: boehm@parc.xerox.com)

SUMMARY

Programming languages generally provide a 'string' or 'text' type to allow manipulation of sequences of characters. This type is usually of crucial importance, since it is normally mentioned in most interfaces between system components. We claim that the traditional implementations of strings, and often the supported functionality, are not well suited to such general-purpose use. They should be confined to applications with specific, and unusual, performance requirements. We present 'ropes' or 'heavyweight' strings as an alternative that, in our experience leads to systems that are more robust, both in functionality and in performance.

Ropes have been in use in the Cedar environment almost since its inception, but this appears to be neither well-known, nor discussed in the literature. The algorithms have been gradually refined. We have also recently built a second similar, but somewhat lighter weight, C-language implementation, which is included in our publically released garbage collector distribution.[1] We describe the algorithms used in both, and give some performance measurements for the C version.

KEY WORDS: character strings; concatenation; Cedar; immutable; C; balanced trees

WHAT'S WRONG WITH STRINGS?

Programming languages such as C and traditional Pascal provide a built-in notion of strings as essentially fixed length arrays of characters. The language itself provides the array primitives for accessing such strings, plus often a collection of library routines for higher level operations such as string concatenation. Thus the implementation is essentially constrained to represent strings as contiguous arrays of characters, with or without additional space for a length, expansion room, etc.

There is no question that such data structures are occasionally appropriate, and that an 'array of characters' data structure should be provided. On the other hand, since the character string type will be used pervasively to communicate between modules of a large system, we desire the following characteristics:

1. Immutable strings, i.e. strings that cannot be modified in place, should be well supported. A procedure should be able to operate on a string it was passed without danger of accidentally modifying the caller's data structures. This becomes particularly important in the presence of concurrency, where in-place updates to strings would often have to be properly synchronized. (Although they are not the norm, immutable strings have been provided by SNOBOL and BASIC since the mid 1960s. Thus this idea is hardly new.)
2. Commonly occurring operations on strings should be efficient. In particular

CCC 0038–0644/95/121315–16

Received 26 July 1994
Revised 7 April 1995

(non-destructive) concatenation of strings and non-destructive substring operations should be fast, and should not require excessive amounts of space.
3. Common string operations should scale to long strings. There should be no practical bound on the length of strings. Performance should remain acceptable for long strings. (All of us have seen symptoms of the violation of this requirement. For instance, the vi editor on most UNIX(TM) systems is unusable on many text files due to a line length limit. An unchecked input limit in fingerd supported the Morris internet worm.[2] A six-month-old child randomly typing at a workstation would routinely crash some older UNIX kernels due to a buffer size limitation, etc.)
4. It should be as easy as possible to treat some other representation of 'sequence of character' (e.g. a file) as a string. Functions on strings should be maximally reusable.

Strings represented as contiguous arrays of characters, as in C or Pascal, violate most of these. Immutable strings may or may not be supported at the language level. Concatenation of two immutable strings often involves copying both, and thus becomes intolerably inefficient, in both time and space, for long strings. The substring operation usually (though not necessarily) exhibits similar problems. Since strings are stored contiguously, any copying of strings results in the allocation of large chunks of storage, which may also result in substantial memory fragmentation. (For long strings, we have observed this to be a problem even for some compacting garbage collectors, since they are likely to avoid moving very large objects.)

As mentioned above, it is very common for application programs not to scale to long string inputs at all. When they do, they commonly use special purpose data structures to represent those strings in order to obtain acceptable performance. We are not aware of any standard UNIX text editors that use a general purpose string representation for the file being edited. Doing so would make character insertion in long files intolerably slow. The approach we propose makes that practical.

In order to maximize reusability of string functions, it should be easy to coerce other representations into standard strings. This is always possible by copying the characters and building the appropriate string representation. But this is undesirable if, for example, the original sequence is a long file, such that only the first few characters are likely to be examined. We would like to be able to convert files into strings without first reading them.

AN ALTERNATIVE

In order to allow concatenation of strings to be efficient in both time and space, it must be possible for the result to share much of the data structure with its arguments. This implies that fully manual storage management (e.g. based on explicit *malloc/free*) is impractical. (It can be argued that this is true even with conventional string representations. Manual storage management typically results in much needless string copying.) Though an explicitly reference counted implementation can be built, we will assume automatic garbage collection.

Since concatenation may not copy its arguments, the natural alternative is to represent such a string as an ordered tree, with each internal node representing the concatenation of its children, and the leaves consisting of *flat* strings, usually rep-

resented as contiguous arrays of characters. Thus the string represented by a tree is the concatenation of its leaves in left-to-right order, as shown in Figure 1.

We refer to character strings represented as a tree of concatenation nodes as *ropes*. (This is a little sloppy. A rope may contain shared subtrees, and is thus really a directed acyclic graph, where the out-edges of each vertex are ordered. We will continue to be sloppy.)

Ropes can be viewed as search trees that are indexed by position. If each vertex contains the length of the string represented by the subtree, then minimal modifications of the search tree algorithms yield the following operations on ropes:

1. *Fetch ith character.* A simple search tree look-up. Rather than examining the subtree containing the right key, we examine the tree containing the proper position, as determined by the length fields.
2. *Concatenate two ropes.* Search tree concatenation as defined in Reference 3.
3. *Substring.* Two search tree split operations, as defined in Reference 3.
4. *Iterate over each character.* Left-to-right tree traversal.

The first three of the above operations can be performed in a time logarithmic in the length of the argument, using, for example, B-trees or AVL trees.[3,4] Note that since strings are immutable, any nodes that would be modified in the standard version of the algorithm, as well as their ancestors, are copied. Only logarithmically many nodes need be copied.

The last can be performed in linear time for essentially any search tree variant. Thus both concatenation and substring operations (other than for very short substrings) are asymptotically faster than for conventional flat strings. The last exhibits roughly the same performance. The first is somewhat slower, but usually infrequent.

In practice, we modify this in two ways. Concatenation is often a sufficiently important operation that it should run in unit, not logarithmic, time. Long output strings are typically built by concatenating short ones. For example, compilers generating assembly language output may do so by concatenating strings in memory.[5] Hence binary concatenation normally simply adds a root node, and does not rebalance the tree. The rebalancing operation is either performed selectively, or invoked explicitly. Effectively this trades speed of the substring and fetch operations for better concatenation performance. Iteration over a rope is basically unaffected.

(Theoretically it is possible to guarantee that the first three operations run in amortized logarithmic time, the iteration operation runs in linear time, and individual

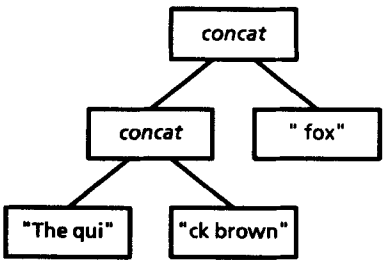

Figure 1. Rope representation of 'The quick brown fox'

concatenations are performed in constant time by using a 'splay tree' based representation,[6,7] in which each node represents the concatenation of a (possibly empty) rope, a (non-empty) flat string, and another rope. This appears impractical in our situation, since the fetch operation requires large amounts of allocation and data structure modification. Furthermore, in a multithreaded environment, the fetch operation appears to require locking.)

It is useful to introduce additional kinds of tree nodes. At a minimum, we allow a second kind of leaf node containing at least a length and a user-defined function for computing the *i*th character in the string. This allows other representations of character sequences (e.g. files) to be treated as ropes without copying. It may further be useful to introduce substring nodes, so that long substrings of flat ropes do not require copying.

ALGORITHMS

We briefly discuss the implementation of the more common operations. We will largely ignore nodes containing user-defined functions, in that they do not affect the algorithms significantly. We assume that all leaves are non-empty. We exclude empty ropes from consideration; they are a trivial special case.

Note that all operations are completely non-destructive. They may be performed without locking in a multithreaded environment.

Concatenation

In the general case, concatenation involves simply allocating a concatenation node containing two pointers to the two arguments. For performance reasons, it is desirable to deal with the common case in which the right argument is a short flat string specially. If both arguments are short leaves, we produce a flat rope (leaf) consisting of the concatenation. This greatly reduces space consumption and traversal times. If the left argument is a concatenation node whose right son is a short leaf, and the right argument is also a short leaf, then we concatenate the two leaves, and then concatenate the result to the left son of the left argument. Together, these two special cases guarantee that if a rope is built by repeatedly concatenating individual characters to its end, we still obtain a rope with leaves of reasonable size. They also greatly reduce the imbalance of the resulting trees.

Since the length of ropes is, in practice, bounded by the word size of the machine, we can place a bound on the depth of balanced ropes. The concatenation operation checks whether the resulting tree significantly exceeds this bound. If so, the rope is explicitly rebalanced (see below). This has several benefits:

1. The balancing operation is only invoked implicitly for long ropes, and then rarely. Each balancing operation will normally reduce the depth of the rope to considerably below the threshold.
2. Recursive operations on ropes require a bounded amount of stack space.
3. Paths in the tree can be represented in a fixed amount of space. This is important for the C implementation (see below).

Substring

The substring operation on structured ropes can be easily implemented. We assume that the substring operation on leaves simply copies the relevant section of the leaf, and deals with negative start arguments and over-length arguments correctly.

```
substr(concat(rope1,rope2),start,len) =
    let
      left = if start ≤ 0 and len ≥ length(rope1) then
               rope1
             else
               substr(rope1,start,len)
      right = if start ≤ length(rope1)
                and start + len ≥ length(rope1) + length(rope2) then
                rope2
              else
                substr(rope2,start-length(rope1), len-length(left))
    in
      concat(left,right)
```

This involves one recursive call for each tree node along the left or right boundary of the substring. Hence its running time is bounded asymptotically by the tree height.

There is a trade-off between this kind of eager substring computation, and one in which substrings are computed lazily by introducing special substring nodes, representing unevaluated substrings. In practice we want to use lazy substring computations at least when we compute long substrings of very long flat ropes (e.g. a function node representing a lazily-read file).

Rebalancing

Rebalancing produces a balanced version of the argument rope. The original is unaffected.

We define the depth of a leaf to be 0, and the depth of a concatenation to be one plus the maximum depth of its children. Let F_n be the nth Fibonacci number. A rope of depth n is balanced if its length is at least F_{n+2}, e.g. a balanced rope of depth 1 must have length at least 2. Note that balanced ropes may contain unbalanced subropes.

The rebalancing operation maintains an ordered sequence of (empty or) balanced ropes, one for each length interval $[F_n, F_{n+1})$, for $n \geq 2$. We traverse the rope from left to right, inserting each leaf into the appropriate sequence position, depending on its length. The concatenation of the sequence of ropes in order of decreasing length is equivalent to the prefix of the rope we have traversed so far. Each new leaf x is inserted into the appropriate entry of the sequence. Assume that x's length is in the interval $[F_n, F_{n+1})$, and thus it should be put in slot n (which also corresponds to maximum depth $n - 2$). If all lower and equal numbered levels are empty, then this can be done directly. If not, then we concatenate ropes in slots $2,\ldots,(n-1)$ (concatenating onto the left), and concatenate x to the right of the result. We then

continue to concatenate ropes from the sequence in increasing order to the left of this result, until the result fits into an empty slot in the sequence.

The concatenation we form in this manner is guaranteed to be balanced. The concatenations formed before the addition of x each have depth at most one more than is warranted by their length. If slot $n - 1$ is empty then the concatenation of shorter ropes has depth at most $n - 3$, so the concatenation with x has depth $n - 2$, and is thus balanced. If slot $n - 1$ is full, then the final depth after adding x may be $n - 1$, but the resulting length is guaranteed to be at least F_{n+1}, again guaranteeing balance. Subsequent concatenations (if any) involve concatenating two balanced ropes with lengths at least F_m and F_{m-1} and producing a rope of depth $m - 1$, which must again be balanced.

Figure 2 shows this algorithm applied to a representation of 'abcdef'. We have shown only slots 2, . . ,5, which correspond to the length intervals [1,2), [2,3), [3,5), and [5,8), and maximum depths 0, 1, 2, and 3, respectively. Note that the original was already balanced by our definition, as is likely to be the case for small examples.

The above argument almost applies when we add a balanced tree b instead of a leaf to the sequence. We may introduce one additional tree level immediately above the insertion point for b, adding one to the depth of the final tree. In practice we insert already balanced trees as a unit, and thus avoid rebuilding the entire tree in response to a rebalancing operation.

The final step in balancing a rope is to concatenate the sequence of ropes in order of increasing size. The resulting rope will not be balanced in the above sense, but its depth will exceed the desired value by at most 2. One additional root node may be introduced by the final concatenation. (Indeed, this is necessary. Consider concatenating a small leaf to a large balanced tree to another small leaf. We must add 2 to the depth of the resulting tree unless we re-examine the balanced tree. Note that only trees that are balanced by our original strict definition are not re-examined.)

Many variations of this approach are possible. Our balance condition was expressed

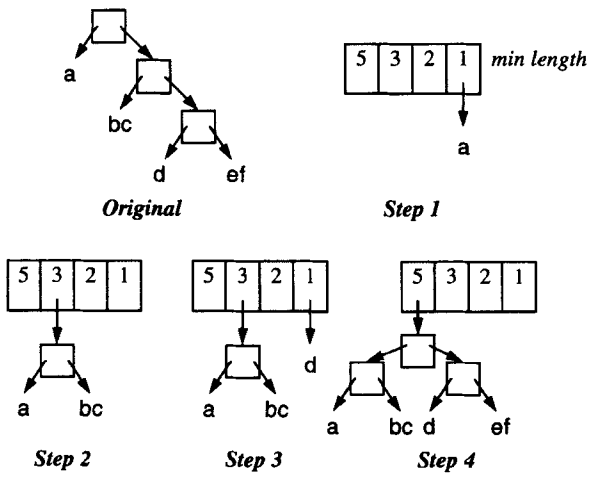

Figure 2. Balancing 'abcdef'

in terms of length, and our algorithm tends to move long flat ropes close to the root. One could also rebalance purely in terms of node count.

C CORDS

We implemented a version of the above as a C library. Since this implementation is lighter-weight than Cedar ropes, and heavier than C strings, we dubbed the resulting objects 'cords'.

The C implementation operates under certain constraints. The compiler dictates that a string constant is represented as a pointer to a NUL-terminated contiguous array of characters. For our package to be practical, C string constants must double as cord constants, whenever possible. Hence cords are also declared to be character pointers ('const char*' to be exact), as are C strings, but are actually represented as either a NULL pointer, denoting the empty cord, or in one of the three ways shown in Figure 3.

The first is a pointer to a *non-empty* C string, the second is essentially a pointer to a function closure for computing the *i*th character in the cord, and the third a pointer to a concatenation node. Concatenation nodes contain a one byte depth field, which is guaranteed to be sufficient, given the word length and balancing constraint. A one byte 'length of left child' field is included, and eliminates nearly

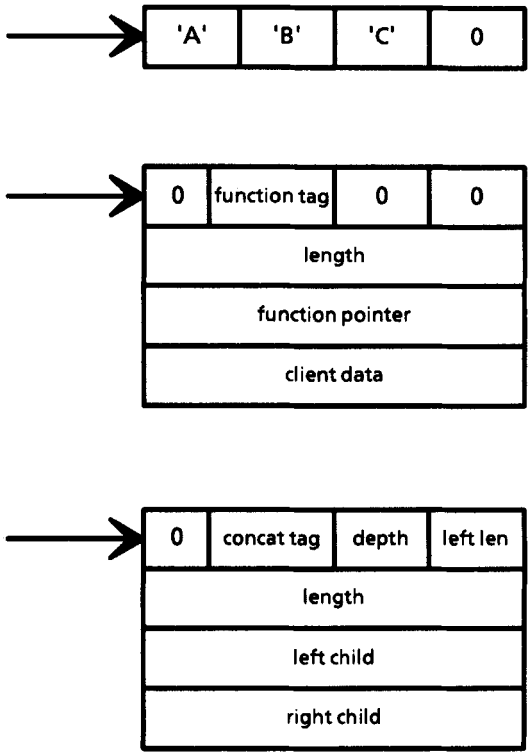

Figure 3. Cord representation variants

all calls to compute the length of C strings during cord traversals. A zero value in this field indicates a long left child.

Cords are balanced exactly as described above. The substring operation may operate lazily and introduce a special variant of a closure node whose environment (pointed to by the client data field) contains a pointer to a (flat) cord, and a starting position within that cord. Substring operations on long flat cords do not copy characters. This special variant is recognized by the substring operation, so that nested substring operations do not result in nested substring nodes.

This design has several benefits:

1. String constants (or other non-empty C strings) may be passed directly to procedures expecting cords. Unfortunately, empty C strings must be converted to a NULL pointer. Erroneous uses of empty C strings could nearly always be cleanly detected, at minimal cost, in a debugging version of the library.
2. Unlike C strings, NUL characters may be embedded inside cords using the closure variant.
3. Concatenation and closure nodes are small.
4. The lengths of non-flat cords may be cheaply computed. Lengths of C strings at leaves may often be cheaply computed from the context in which they appear. (For example, the substring operation can usually do so.)

We use the concatenation optimizations described above. Ropes are balanced either in response to an explicit call, or when the depth of a rope exceeds a threshold larger than the depth of any balanced rope.

Since C does not provide function closures directly, it is clumsy to write client code that invokes an iterator function to traverse a string, tempting programmers to use the significantly slower fetch operation instead. (The recursive version of this operation is even slower than one might think on register window machines.) We sidestep this issue by introducing a *cord position* abstraction. Abstractly, this is a pair consisting of a cord, and a character index in the cord. Concretely, a cord position is represented by these, plus the path from the root of the cord to the leaf containing the index, and some cached information about the current leaf. Cord positions provide fast operations for retrieving the current character, and advancing the position to the next (or previous) character.

The implementation uses a recent version of the conservative garbage collector described in References 8 and 9, as does the current Cedar implementation. It is available by anonymous ftp, together with the garbage collector, and a toy text editor built on cords.[1] This editor maintains a complete edit history as simply a stack of cords, each cord representing the entire file contents. Performance is usually more than adequate, even on multi-megabyte files. Memory use is moderate, both since history entries share nearly all of their space, and because large files can be read lazily, and not kept in memory. This implementation is, however, not nearly as mature as the Cedar implementation.

CEDAR ROPES

Cedar ropes are explicitly supported by the language. The representation of normal flat ropes (leaves) coincides with that of a built-in flat string type 'text'. String constants have the type 'rope' or 'text', depending on context.

Concatenation is implemented with the optimizations described above. Balancing is also similar to what is described above.

In addition to the representation variants used for cords, a Cedar rope may be represented by a replacement node, denoting a lazy replacement of a substring of a rope by another rope. It is unclear that this variant is necessary, since it could also be represented by a concatenation with substrings.

Instead of leaves containing user-defined functions, Cedar ropes may be represented by 'object' nodes that contain 'member' functions to perform several different operations, including fetching a specific character. (All of the operations other than 'fetch' have defaults provided.) This makes such nodes a bit more expensive to manipulate, but makes certain operations (e.g. traversal and copying characters to a flat buffer) considerably cheaper.

Cedar substring nodes are completely separate from function nodes, and must be treated distinctly. As a result, they can be, and are, more heavily used. The substring operation descends into the tree until it finds the smallest rope r that entirely contains the desired substring. It then returns a substring node that refers to r.

Rope traversal is accomplished primarily with a mapping function that applies a user-defined function to each character. Since Cedar allows nested functions to be passed as arguments, this provides a much cleaner interface than in C.

The vast majority of interfaces that deal with sequences of characters traffic in ropes. Rope nodes are the single most commonly occurring object in most Cedar heaps, both in terms of their number and their total size.

Much of the Cedar environment is available as Reference 10.

PERFORMANCE

The Cedar system makes heavy use of ropes throughout. The text editor represents an unstructured text file as a single rope. The performance of the system is quite competitive.

Similarly, the above-mentioned toy editor performs quite well, even on very large files. In order to give some more quantitative comparisons of the techniques we advocate with conventional string programming techniques, we measured operations likely to be common in either case.

Figure 4 reports the cost of string concatenations in CPU microseconds on a Sun SPARCstation 2. We measured the cost of concatenating two strings or ropes of various lengths. The top three lines reflect the cost of non-destructive concatenation using the standard C library with several different memory allocators. The bottom two correspond to rope concatenation, again with slightly different allocation overhead. More detailed explanations of all the measurements are deferred to the appendix.

Figure 5 gives the time required to build a single C cord or conventional C string a character at a time, using various common, though not necessarily good, programming techniques. These correspond roughly to the techniques that might be used when reading in a text file or a command line, and buiding an in-memory version of the string. The vertical axis measures the time required to build a string of the indicated length. Note that not all operations are strictly comparable in functionality.

The lines labelled UNBUFFERED and BUFFERED were obtained using the standard

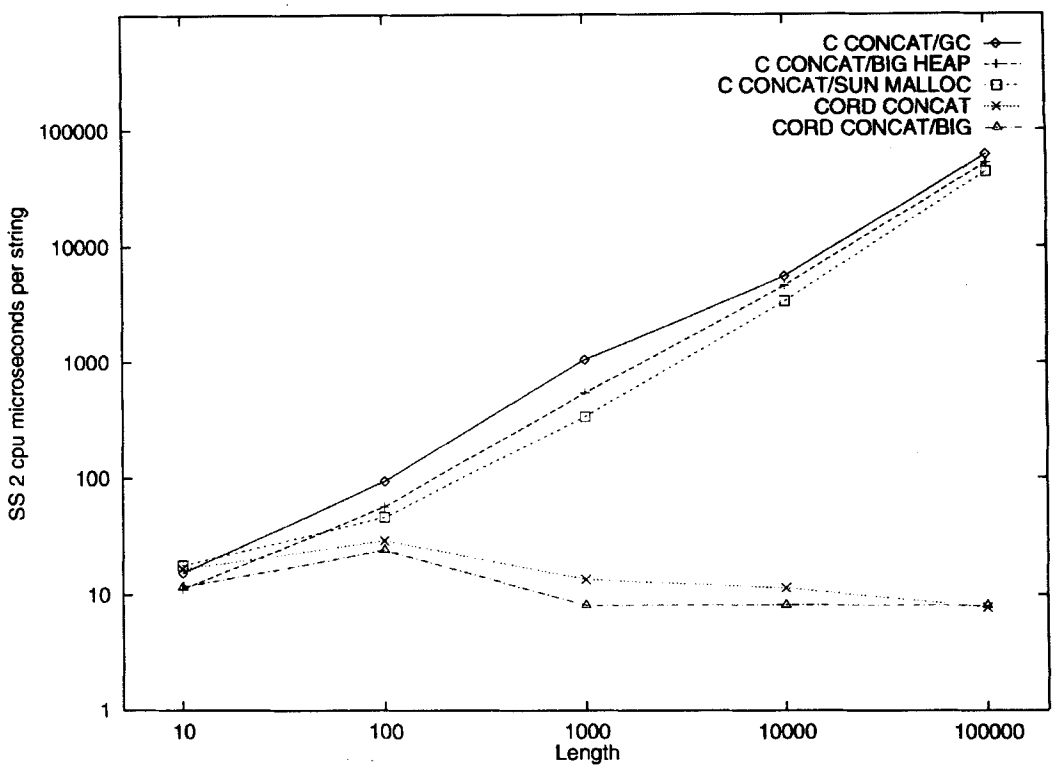

Figure 4. Concatenation time

C realloc and strcat functions to build up the string either 1 or 128 characters at a time. The line labelled SIMPLE builds up the string in place (which is fastest, but not always practical). GEOMETRIC does the same, but starts with a small buffer and repeatedly doubles it. The remaining two lines use the CORD package, either by naïvely concatenating individual characters to a string (CAT BUILD), or by using an interface explicitly designed for the purpose (BUFFERED CORD). Note that BUFFERED CORD provides essentially the same performance as GEOMETRIC, and that CAT BUILD might still be tolerable for 100,000 character strings, where UNBUFFERED is completely unacceptable.

Finally, Figure 6 gives the time required to completely traverse a C style string or cord of a given length. Times are again given per string traversal. The top four lines represent cord traversals, with varying programming sophistication. The bottom line represents a simple C traversal of a contiguous string of the same length.

PREVIOUS WORK

Many imperative programming languages have recognized the desirability of an immutable string type. The earliest language with immutable strings that we could find was SNOBOL,[11,12] followed by BASIC and XPL. SNOBOL implementations included flat immutable strings, together with a fast substring operation, and provisions for fast concatenation in special cases. The same is true of SAIL[13] and a

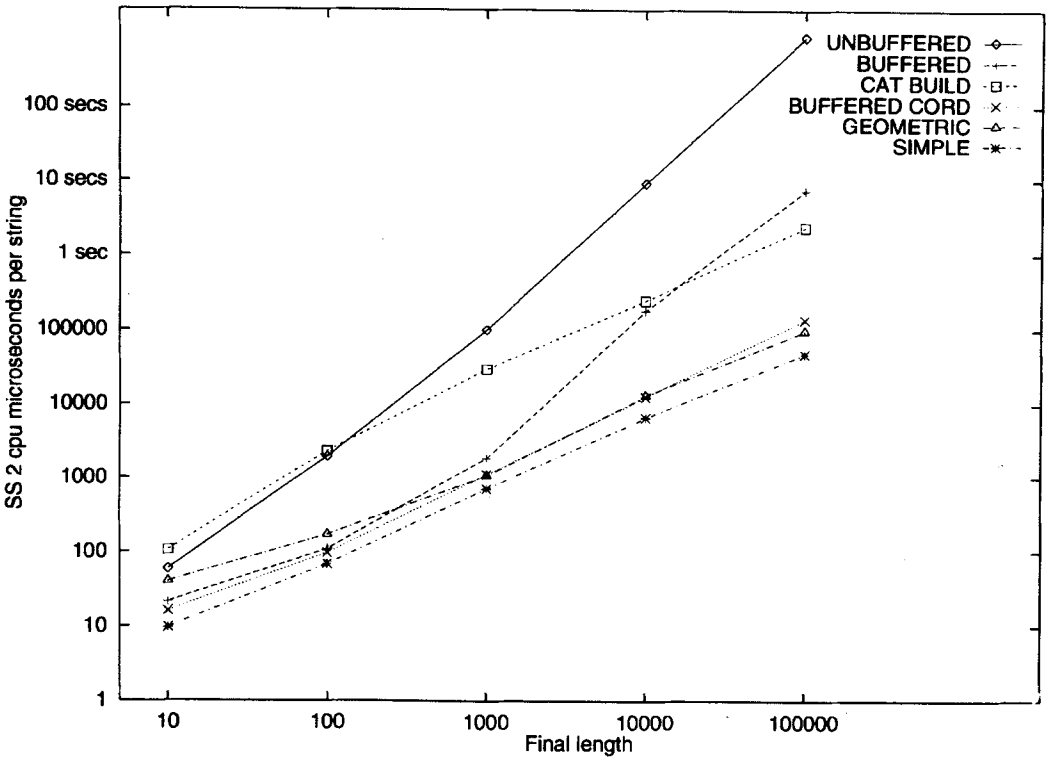

Figure 5. Build time

number of other later language designs such as CLU,[14] Icon[15] and standard ML. The CLU design directly inspired Cedar ropes. But all implementations of which we are aware use flat representations, and hence cannot, for example, efficiently insert a character into the middle of a 100,000 character string.

Linked string representations are considered in Reference 16. However, its author considers only linearly linked representations designed for fast in-place updates. Non-destructive concatenation and substring operations perform poorly on these representations.

Several authors have discussed general techniques for efficient implementations of immutable data structures. Most of these rely on the use of balanced 'tree' structures, and copying of modified nodes, as we do. Probably the closest to our work is Reference 17, which discusses non-destructive operations on linear lists using AVL DAGs. The string case is not specifically discussed (though an editor application is also used). Our data structure is quite different from that of Reference 17 to accommodate fast concatenation, the small list elements (characters), lazily evaluated sublists, and the need for proper integration with string constants.

A more sophisticated implementation technique for immutable data structures is described in Reference 18. That approach could be used, for example, to implement ropes with $O(1)$ space cost for character insertion and deletion, even when the original rope is retained. However their approach again appears to require locking.

The idea of representing a string as a tree of concatenations has been rediscovered

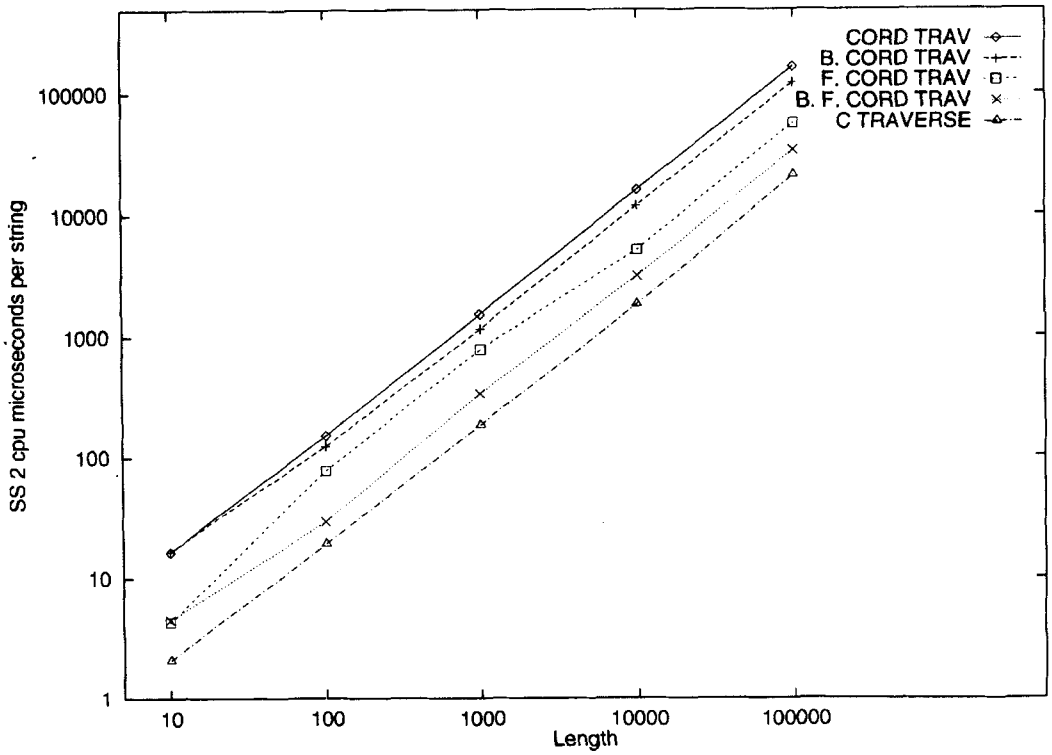

Figure 6. Traversal time

multiple times, particularly in the context of attribute grammars (cf. Reference 5 or Rodney Farrow's Linguist system). Here a fast concatenation operation is often sufficient. We do not know of other published generalizations to a comprehensive collection of string operations.

Cedar Ropes themselves have evolved since 1982. They are mentioned in passing, but not described, in References 19 and 20. References 21 and 22 each contain a one paragraph description of ropes, with no implementation details (some of which have changed since then). Reference 21 also discusses some applications of ropes.

Variations of ropes have also appeared in DEC SRC programming environments. Modula-2+ provided both an interface and implementation similar to Cedar ropes. Modula 3^{23} provides the interfaces, but the current implementation uses a flat representation. (Though we have not done a thorough study, this choice appears to at least double the heap space requirements of one of the Modula-3 programs we did examine.)

CONCLUSIONS

We have argued that it is important to provide an immutable character string type that provides a cheap concatenation operation, and scales correctly to long strings, so that modules with string inputs can be truly reusable. It is important that programming languages either provide this facility or can be easily extended to provide it.

The Cedar language supports it explicitly. It is possible to add it to C as a library, with some small further loss of type safety. This is much more than compensated for by the elimination of subscript errors resulting from needless low-level string manipulation.

It is not clear whether the implementation we suggested above is in any sense optimal. However, it performs sufficiently well for our purposes, and sufficiently well that for most purposes it is preferred over traditional 'flat' string representations. Our experience has been that it results in software that is noticeably more robust, particularly in that it scales correctly to unexpectedly large inputs.

ACKNOWLEDGEMENTS

Many other members of PARC CSL have contributed to the Cedar Rope package. Dan Swinehart and one of the referees helped us track down the history of immutable strings. Both anonymous referees provided useful suggestions on an earlier draft of this paper.

APPENDIX

All measurements in Figures 4, 5, and 6 are in user mode CPU microseconds on a SPARCstation 2 running running SunOS 4.1.2. We used version 4.1 of the package in Reference 1. The machine had enough memory (64 MB) that paging was not an issue. All programs were compiled with gcc and statically linked. Garbage collection times are included. The tests were run for enough iterations in the same heap to force several collections.

Though the measurements are quite repeatable, they should be taken as rough values, since they depend on many parameters not discussed here. (For instance, the pointers to the strings were kept in statically-allocated global variables.)

Figure 4

C CONCAT/GC

Non-destructively concatenate a string to itself, using two strlen calls, an allocation, and two memcpy calls. This uses version 4.1 of our conservative garbage collector in its default (more space- than time-efficient) configuation. (The contiguous strings of length 200,000 were allocated with interior pointer recognition disabled to avoid anomalous behavior. See Reference 9 for a discussion of why this is desirable.)

C CONCAT/BIG HEAP

As above, but the heap was expanded by 4 MB at start-up. This significantly reduces garbage collection overhead.

C CONCAT/SUN MALLOC

As above, but uses the SunOS 4.1.3 default malloc. Not that this actually performs slightly worse than even C CONCAT/GC for short strings, but outperforms the garbage collected versions for long strings (which force more frequent collection).

CORD CONCAT

Concatenate a cord to itself. Uses the garbage collector in default configuration. The cord was built as in BUFFERED CORD below. This matters only in that the cord is structured, and hence has an easily available length, except in the length 10 and 100 cases. The time per operation is basically constant, except that short strings require calls to strlen, and that long strings force a large heap, which tends to reduce GC overhead.

CORD CONCAT/BIG

Same as CORD CONCAT, except that the heap was expanded by 4 MB at start-up.

Figure 5

All measurements were performed with our garbage collecting allocator. Our version of C's realloc explicitly deallocates memory; no other storage was explicitly deallocated.

UNBUFFERED

Build up a C string by initially allocating a single byte, and then repeatedly reallocating the buffer and adding the new character to the end with the C strcat function. Note that this copies the string only when realloc copies. (The realloc implementation copies at geometrically increasing lengths for small strings, and whenever the new object requires an additional page for large strings.) Every character involves a traversal of the first string. (This is a highly suboptimal algorithm, which unfortunately appears to be used occasionally.)

BUFFERED

Build up a C string by initially allocating a single byte, putting new characters in a buffer, and destructively appending the buffer to the strings when it is full. Append is implemented using C's realloc and strcat. The buffer size is 128 bytes.

CAT BUILD

We build a cord. A cord consisting of a single character is repeatedly concatenated to the cord. (This is usually an amateur programming technique, roughly comparable to UNBUFFERED. It does have the advantage that nothing is being destructively updated. Any prefix can be safely saved with a pointer copy.)

BUFFERED CORD

Similar to BUFFERED, but each time the buffer fills, it is concatenated to the cord. This uses routines provided by the cord implementation for this purpose. The buffer size is again 128. The resulting cord is explicitly balanced. The interface is similar to Modula-3 'text writers'.[23] The implementation is tuned for short strings.

GEOMETRIC

The space allocated to the string is grown by factors of 2, starting at 1. Strcat is not used, string length is not recomputed.

SIMPLE

Build up a C string by preallocating a sufficiently large buffer and filling it in. (A commonly used technique. Fails for unexpectedly long strings.)

Figure 6

The traversal function computes the exclusive-or of all characters in the string. Thus the compiler is prevented from optimizing out major parts of the traversal.

CORD TRAV

Traverse a cord built by CONCAT, using the cord position primitives described above. With the help of some macros provided by the cord package, the traversal code looks very similar to what is needed for traversing a traditional C string.

B. CORD TRAV

Same as CORD TRAV, but uses the cord from BUFFERED CORD. The cord is more balanced with larger leaves.

F. CORD TRAV

Same as CORD TRAV, but does not use the cord position primitives. Instead it explicitly invokes an iterator function. The iterator function is passed a function that examines an individual character. It can also be optionally passed a second function that will accept longer substrings (as C strings) and process more than one character at a time. In this case such a function was also provided. This takes about 15 more lines of code than CORD TRAV. (This code would have to be repeated for each type of string traversal.)

B.F. CORD TRAV

Same as F. CORD TRAV, but uses the cord from BUFFERED CORD. This represents the best case for a cord traversal.

C TRAVERSE

Traverse a C string using explicit inline array subscripting. The C string pointer was stored in a global, but the resulting load was optimized out by the compiler.

REFERENCES

1. The garbage collector and cord package can be retrieved from ftp://parcftp.xerox.com/pub/gc/gc.tar.Z. A brief description can be found in ftp://parcftp.xerox.com/pub/gc/gc.html.

2. Eugene H. Spafford, 'Crisis and aftermath', in 'Special Section on the Internet Worm', *Communications of the ACM* **32**, (6), 678–687, (1989).
3. D. Knuth, *The Art of Computer Programming, Vol. 3, Searching and Sorting*, Addision-Wesley, 1973.
4. Samuel W. Bent, Daniel D. Sleator and Robert E. Tarjan, 'Biased 2-3 trees', *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, IEEE, 1980, pp. 248–254.
5. H. Boehm and W. Zwaenepoel, 'Parallel attribute grammar evaluation', *Proceedings of the 7th International Conference on Distributed Computing Systems*, September 1987, pp. 347–355.
6. Robert R. Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.
7. D. Sleator and R. Tarjan, 'Self-adjusting binary trees', *Proceedings, Fifteenth Annual ACM Symposium on Theory of Computing*, 1983, pp. 235–245.
8. Hans-J. Boehm and Mark Weiser, 'Garbage collection in an uncooperative environment', *Software—Practice and Experience* **18**, 807–820 (1988).
9. Hans-J. Boehm, 'Space efficient conservative garbage collection', *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, SIGPLAN Notices* **28**, (6), 197–206 (1993).
10. C. Hauser, C. Jacobi, M. Theimer, B. Welch and M. Weiser, 'Using threads in interactive systems: a case study', *Xerox PARC Technical Report CSL-93-16 CD*. This is a CD-ROM containing both a paper and a Cedar system that can be run under SunOS. Source code is included for much of the system, including the Rope package.
11. D. Farber, R. E. Griswold and I. P. Polansky, 'SNOBOL, a string manipulation language', *Journal of the Association for Computing Machinery*, **11**, (1), 21–30 (1964).
12. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, Second Edition, Prentice-Hall, 1971.
13. Kurt A. VanLehn (ed.), 'SAIL user manual', *Report STAN-CS-73-373*, Stanford Computer Science Department, July 1973.
14. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheiffler and A. Snyder, *CLU Reference Manual*, Springer, 1981.
15. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
16. Stuart E. Madnick, 'String processing techniques', *Communications of the ACM*, **10**, (7), 420–424, (1967).
17. Eugene W. Myers, 'Efficient applicative data types', *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, January 1984, pp. 66–75.
18. D. Driscoll, N. Sarnak, D. Sleator and R. Tarjan, 'Making data structures persistent', *JCSS*, **38**, (1), 86–124 (1989).
19. Butler W. Lampson, 'A description of the Cedar programming language: a Cedar language reference manual', *Xerox PARC Report CSL 83–15*, December 1983 (out of print).
20. D. Swinehart, P. Zellweger and R. Hagmann, 'The structure of Cedar', *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments, SIGPLAN Notices*, **20**, (7), 230–244 (1985).
21. D. Swinehart, P. Zellweger, R. Beach and R. Hagmann, 'A structural view of the Cedar programming environment', *Xerox PARC Report CSL 86.1*, June 1986.
22. W. Teitelman, 'A tour through Cedar', *IEEE Software*, April 1984, pp. 44–73.
23. Greg Nelson (ed.), *Systems Programming in Modula-3*, Prentice Hall, 1991.