Efficient Representations for Large Dynamic Sequences in ML

Arthur Charguéraud, Mike Rainey

Efficient Representations for Large Dynamic Sequences in ML

Arthur Charguéraud
Inria
arthur.chargueraud@inria.fr

Mike Rainey
Inria
mike.rainey@inria.fr

Growable containers

When the maximal number of elements that may be inserted into a container, such as a stack, a queue or a double-ended queue, is not known in advance, the container needs to grow dynamically. Growable containers are most frequently implemented on top of functional or imperative lists, or on top of vectors (i.e., resizable arrays). Yet, lists and vectors are inefficient in both space and time.

Lists require the allocation of a three-word object for each item. Furthermore, traversing lists involves numerous indirections, slowing down iteration and GC processing. Other tree structures that store elements in leaves typically suffer from the same drawbacks.

Vectors typically require two words per item due to the resizing strategy. Worse, upon shrinking, resizing typically takes place when the vector becomes quarter full. As such, it requires up to four words per item. (Other ratios are possible, but they lead to poorer performance.) Furthermore, vector resizing operations have a cost that, while amortized, nevertheless adds to the constant factors. The cost of copying is especially visible for long sequences, for which the copy operation triggers numerous cache misses.

Since growable containers are ubiquitous in programming, there is ample motivation to seek space-efficient (and time-efficient) representations of growable containers. Signifiant gains may be expected for containers storing large numbers of elements.

Chunks: towards compact representations

One classic approach to obtaining compact (space-efficient) representations of dynamic sequence data structures (stacks, queues, double-ended queues, and strings) is to store elements in *chunks*, where each chunk is essentially a fixed-capacity array. The exact representation depends on the set of operations supported, and on whether the data structure is ephemeral (i.e. with destructive updates) or fully-persistent (i.e. similar to a purely-functional structure).

For example, an ephemeral chunk supporting stack operations may be represented as a record made of an array of size K and an integer storing the number of elements stored in the chunk. For example, a persistent chunk supporting stack operations may be represented efficiently by exploiting sharing when possible. We represent a persistent chunk as a pointer to an ephemeral chunk, called the *support*, and an integer representing the size of the persistent chunk. The idea is that the persistent chunk is a subsequence of its support. The support may be shared by several persistent chunks: by the results of pop operations and the results of push operations that increase the size of the support for the first time. (Other push operations require a full copy of the support chunk.) Details are given in the appendix.

Typically, all chunks but the ones at the ends of the sequence are full. Thus, the only memory overhead consists of these chunks, plus a constant overhead per chunk, for its header and for the linking between the chunks. Overall, the total space usage for a sequence of n single-word items stored in chunks of capacity K is of the form $(1 + O(1)/K)n + O(K)$ words. For $K \geq 128$ and non-tiny sequences, we require roughly $1.1n$ words. This bound is clearly much better than $3n$ (for lists), or between $2n$ and $4n$ (for vectors).

In addition to space-efficiency, chunk-based data structures bring additional significant benefits: (1) an element is written exactly once into a chunk, and thereafter never needs to be moved (unlike with vector resizing operations), thus saving on the number of memory operations required, (2) iteration over the elements benefits from good locality compared with lists or trees, (3) the use of chunks eases the work of the GC: faster traversal of data, and allocation of small blocks, always of the same size.

In what remains, we present two benchmark results. Our aim is not to make general conclusions, but simply to convince the reader that, beyond the theoretical benefits of chunk-based data structures in terms of space efficiency, such structures may deliver optimal performance time-wise in a number of practical applications.

Benchmark on LIFO operations

We benchmark the following stack data structures:

- A non-resizable array (called `stack_array`), of size `length` (a parameter that varies in the experiment), paired in a record with an integer that keeps track of the size of the stack.
- A reference on a purely functional list of elements.
- A vector, implemented as a record of an array and an integer keeping track of the size. When the array is full, its size gets doubled, when the array is less than 1/4 full, its size gets halved.
- Our ephemeral chunked stack, which is made of one partially-filled chunk and a purely functional list of full chunks. For the chunk capacity, we use $K = 256$, as preliminary investigations show that larger values do not reduce the overheads any further.
- A reference on our persistent chunked stack, which is like its ephemeral counterpart, but uses persistent chunks.

Our test program repeats the following sequence of operations: push `length` items into the stack, then pop all these items. It stops when 40 million operations (20m push and 20m pop) are performed. Figure 1 shows results for various values of the `length` parameter.

- The non-resizable array is usually the fastest structure, although it is slightly outperformed by lists for short sequences (because allocation is sometimes cheaper than mutation in OCaml).
- The list-based stack delivers best performance for stacks of up to 10k elements; however, it becomes the slowest of all the benchmarked structures for stacks of more than 100k elements.

ML Workshop '17 XXX

ACM XXX…$15.00

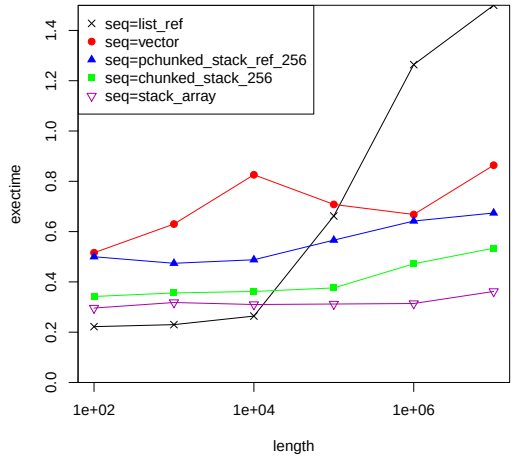

Figure 1. LIFO experiment. Repeatedly creating stacks of `length` items, until 40 million operations are performed in total.

- The vector-based stack is always quite slow, between 2x and 3x slower than a non-resizable array. This result is not so surprising after all, given the need to pay for all the resizing operations. (The irregularity of the curve is a consequence of the fact that input sizes are not powers of 2.)
- The ephemeral chunked stack is at most 1.5x slower than the non-resizable array.
- The persistent chunked stack is at most 2x slower than the non-resizable array. (Note: this benchmark makes single-threaded use of the stack, and thus never triggers any chunk copy.)

Benchmark on string buffer operations

We specialize our stack data structure to implement an extensible string interface. We consider not just an ephemeral string buffer, but also a persistent string buffer: any string version that is created during the process remains forever a valid description of that string.

We benchmark it against the `Buffer` module from OCaml's standard library. The buffer, initially empty, gets appended with words of various sizes until its size grows to 1 billion. The words that are appended are preallocated in the beginning, and they have length between one and `max_word_length` (a parameter that varies in the experiment). There is exactly one word of each length, and these words are appended to the buffer in round-robin fashion.

As shown by the results from Figure 2, our ephemeral chunked string buffer is competitive with (or outperforms) the buffer from the standard library. Our persistent buffers are almost as efficient as our ephemeral buffers. As expected, they are slightly slower when the words pushed are really small (on average 25 characters or fewer), but not by so much.

Concatenation, splitting and random access

To support push and pop operations at either just one or both ends of a sequence, it suffices to maintain a linked list between the chunks that make up the sequence. However, to support more advanced sequence operations, such as concatenation, splitting, or random access, one needs to introduce a fancier tree structure in which to store the chunks. In practice, we may use a finger tree (Hinze and Paterson 2006), or a bootstrapped chunked sequence (Acar et al. 2014), a simpler structure with fairly similar bounds in practice.

We have already implemented concatenation and splitting for ephemeral sequences. We are currently working on supporting these operations on persistent sequences. One particularly promising application is the design of fully-persistent strings with support for concatenation and substring operations, both in logarithmic time.

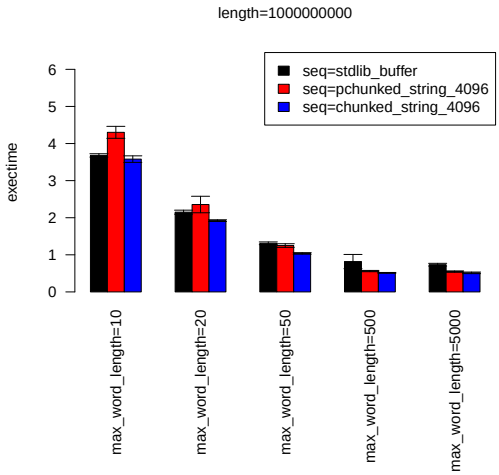

Figure 2. String buffer experiment. Words, each of length between 1 and `max_word_length`, are repeated added to the buffer, until the buffer contains 1 billion characters.

Ongoing work includes: (1) support for efficient iterators, including random-access iterators; (2) development of adaptative variants of our structures to provide space-efficient representation of short sequences; (3) investigation of alternative methods to deal with persistence, following the approach of *transient* structures popular in the Clojure language, whereby the user may dynamically switch between an ephemeral view and a persistent view, to leverage the benefits of both; and (4) more extensive benchmarking, in particular on real-life programs.

Related work

There exists a representation of resizable arrays that guarantees at most $\sqrt{n}$ wasted space, and that supports random-access in at most two indirections (Brodnik et al. 1999). However, this structure does not seem to generalize to queue operations, and it cannot support concatenation or splitting.

Finger trees (Hinze and Paterson 2006) provide purely-functional sequences with asymptotics that provide push and pop at both ends in amortized constant time and split and concat in logarithmic time, yet with poor constant factors.

Haskell's *yi* package (et al 2010) implements strings by instantiating a finger-tree with chunks of characters in the leaves. However, iterated concatenation of tiny strings might result in very problematic situations, e.g. where each chunk of capacity 512 characters stores only 2 characters (see (Acar et al. 2014), p4).

Chunked sequences (Acar et al. 2014) provide ephemeral sequences, with similar asymptotics as finger trees, but with excellent constant factors. We have adapted chunked sequences to derive persistent sequences. In worst-case scenarios with persistence, push and pop operations might degrade to logarithmic time. This is not much of an issue beacuse in chunked sequences the log is in base K, and $\log_K n \leq 7$ for all practical purposes.

Relaxed Radix Balanced (RRB) vectors (Stucki et al. 2015) provide a representation of persistent sequences. The authors report performance results for a Scala implementation. However, we are not aware of any formal analysis of the space overheads and number of allocations involved with this data structure. A more detailed comparison is a matter for future work.

Early work on Lisp systems proposes the technique of *cdr-coding* to optimize the use singly linked lists by coalescing chunks on the fly (Deutsch 1973). More recent work proposes a compiler optimization that performs a similar optimization for ML-like languages (Shao et al. 1994).

Acknowledgements

This research was partially supported by the French National Research Agency (ANR) under the grant ANR-15-CE25-0008, and partially supported by European Research Council grant ERC-2012-StG-308246.

References

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Theory and practice of chunked sequences. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 25–36, 2014. URL `http://dx.doi.org/10.1007/978-3-662-44777-2_3`.

Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Ian Munro, and Robert Sedgewick. *Resizable Arrays in Optimal Time and Space*, pages 37–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48447-9. doi: 10.1007/3-540-48447-7_4. URL `http://dx.doi.org/10.1007/3-540-48447-7_4`.

L. Peter Deutsch. A lisp machine with very compact programs. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 697–703, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1624775.1624860`.

Jean-Philippe Bernardy et al. The Haskell `yi` package, 2010. `https://github.com/yi-editor/yi`.

Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, March 2006. ISSN 0956-7968. doi: 10.1017/S0956796805005769. URL `http://dx.doi.org/10.1017/S0956796805005769`.

Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 185–195, New York, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.182453. URL `http://doi.acm.org/10.1145/182409.182453`.

Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. RRB vector: A practical general purpose immutable sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 342–354, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784739. URL `http://doi.acm.org/10.1145/2784731.2784739`.

Appendix

Representation of ephemeral chunked stacks

```
type 'a chunk = {
  data : 'a array;
  mutable size : int; }
  (* [0 <= size <= data.length = capacity] *)

type 'a stack = {
   mutable head : 'a chunk;
   mutable tail : ('a chunk) list; }
```

To support both an efficient length-query operation and a random-access operation that requires just two indirections, we may change the tail representation from a list of chunks to a vector of chunks. This change makes the structure more generally useful, but does so at the cost of only a tiny amount of space and time overhead.

Our actual implementation includes a secondary head chunk. This chunk is always either empty of full. Its role is to protect against worst-case scenarios of alternating push/pop operations that might otherwise repeatedly trigger a chunk allocation immediately followed by its deallocation. For use cases where worst-case patterns are unlikely to happen, one may use the simplified representation shown above.

Representation of persistent chunked stacks

```
type 'a pchunk = {
  support : 'a chunk; (* support may be shared *)
  size : int; }
  (* [0 <= size <= support.size, with items from
     support.data.(0) to support.data.(size-1). *)

type 'a pstack = {
   head : 'a pchunk;
   tail : ('a pchunk) list; }
```

For a push operation, there are two cases. If `size` is equal to `support.size`, then we write the new element in the `support.data` array at index `size`. (Indeed, the corresponding slot is previously unused.) We then increment `support.size`, and return a chunk object with the same support and a size field increased by one unit. Otherwise, we need a copy-on-write operation: we allocate a fresh support object with a fresh array, and we copy the relevant elements. This case is more expensive, but is less frequent in most applications.

For a pop operation, the support may always be shared: we simply allocate chunk object with the same support and a size field decreased by one unit.

Experimental setup

Executed on a 64bit Intel Core i7-4600U CPU running at 2.10GHz (4 cores, only 1 used), with 8Gb RAM, 4MB L3 cache, 256kb L2 cache, 32kb L1 cache. OS is Ubuntu 14.04.

Compiled with OCaml v4.04.0, with flambda support for inlining (using `ocamlopt -O2 -unsafe -noassert`).

Each time measure is averaged over 5 runs.