Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays

Phil Bagwell

Since its inception Functional Programming, J. McCarthy, has almost universally used the Linked List as the underpinning data structure. This paper introduces a new data structure, the VList, that is compact, thread safe and significantly faster to use than Linked Lists for nearly all list operations. Space usage can be reduced by 50% to 90% and in typical list operations speed improved by factors ranging from 4 to 20 or more. Some important operations such as indexing and length are typically changed from $O(N)$ to $O(1)$ and $O(lgN)$ respectively. A language interpreter Visp, using a dialect of Common Lisp, has been implemented using VLists and the benchmark comparison with OCAML reported. It is also shown how to adapt the structure to create variable length arrays, persistent deques and functional hash tables. The VArray requires no resize copying and has an average O(1) random access time. Comparisons are made with previous resizable one dimensional arrays, Hash Array Trees (HAT) Sitarski [1996], and Brodnik, Carlsson, Demaine, Munro, and Sedgewick [1999].

Categories and Subject Descriptors: H.4.m [**Information Systems**]: Miscellaneous

General Terms: functional, data-structures

Additional Key Words and Phrases: HAT, Arrays, Resizable Arrays, Linked Lists, Functional Lists, Deques, Hash-Lists

1. INTRODUCTION

From the inception of Functional Programming by J. McCarthy, derived from Church's lambda calculus, the Linked List has formed the underlying data structure. Today this implicit assumption remains and is manifested by the recursive type definition in the design of many modern functional languages. Although the Link List has proven to be a versatile list structure it does have limitations that encourage complementary structures, such as strings and arrays, to be used too. These have been employed to achieve space efficient representation of character lists or provide structures that support rapid random access but they do necessitate additional special operators and lead to some loss of uniformity. Further, operations that require working from the right to left in lists, *foldr* or *merge* for example, must do so using recursion. This often leads to stack overflow problems with large lists when optimizers fail to identify and eliminate tail recursion.

In the 1970's *cdr-coding* was developed to allow a *cons* cell to follow the *car* of the first. Greenblatt [1977], Hansen [1969], Clark and Green [1977], Li and Hudak [1986], Clark [1976], [Bobrow and Clark 1979]. Flag bits were used to indicate the list status. More recently this idea was extended to allow k cells to follow the initial *car*, typically $k = 3$ *to* 7 and compiler analysis used to avoid most run-time flag checking. Shao, Reppy, and Appel [1994], [Shao 1994]. A different approach,

Address: Es Grands Champs, 1195-Dully, Switzerland

based on a binary tree representation of a list structure, has been used to create functional random access lists based to give a $lg\ N$ indexing cost yet still maintain constant head and tail times. Okasaki [1995] Similar concepts together with the new technique of recursive slow down give functional deques with constant insert and remove times. Kaplan and Tarjan [1995]

In this paper an alternative structure, the VList, is introduced combining the extensibility of a Linked List with the random access of an Array. It will be shown that lists can be built to have typically O(1) random element access time and a small, almost constant, space overhead.

To verify that this new structure could in fact form the basis for general list manipulations, such as *cons*, *car* and *cdr*, in a real language implementation, an experimental VList Lisp interpreter (Visp) was created. Based on a dialect of Common Lisp Visp was used to both test list performance and ensure there were no implementation snags through each stage from VList creation to garbage collection. The basic VList structure was adapted to support the common atomic data types, character, integer, float, sub-list and so on. Efficient garbage collection was a significant consideration. Visp then provided a simple framework for benchmarking typical list operations against other functional languages. Performance comparisons were made with the well known and respected implementation of OCAML and are to be found in Section 2.6. Interpreted list manipulation programs in Visp can execute a factor of four or more than their equivalent native compiled versions in OCAML. OCAML on the other hand has a clear advantage for other operations where Visps dynamic scoping and type checking degrade its performance.

Interestingly, the principles used to develop VLists also yield potential solutions to other important problems, that of functional hash-lists, section 3, variable size arrays, section 4, and persistent deques, section 5. This implementation of variable sized arrays, the VArray, builds on and is compared with the previous work on variable arrays HATS by Sitarski [1996], Resizable Arrays in Optimal Time and Space (RAOTS), by Brodnik, Carlsson, Demaine, Munro, and Sedgewick [1999]. The VArray is unique in that it allows growth without data structure copying and yet has a constant average random access time.

It is fascinating to find that such a simple concept can have such a varied application.

2. THE VLIST

2.1 The Concept

A VList is based on the simple notion of creating a linked set of memory blocks where the size of each successive block grows by a factor 1/r to form a geometric series with ratio r, see Fig 1. The list is referenced by a pointer to the base of the last added block together with an offset to the last added entry in that block. At the base of each block a block descriptor contains a link to the previous smaller block Base-Offset, the Size of the block and the offset of the last used location in the block, LastUsed.

All list manipulations can be considered to be constructed from the usual three special functions *cdr*, *cons* and *car*. Given the VList structure, *cdr* is accomplished by simply decrementing the offset part of the pointer. When this becomes zero, at

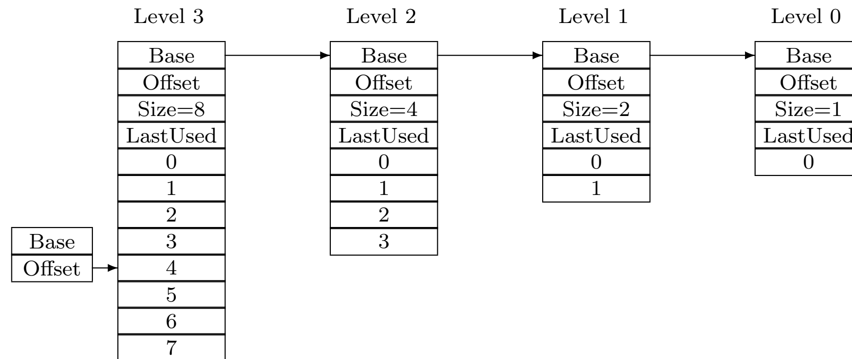

Fig. 1. A VList Structure

a block boundary, the link to the next block Base-Offset is followed and the process continued. While *car* becomes an indirect load via the list pointer.

The list constructor *cons* requires a little more consideration. In Fig 2 a list has been created with the integers (8,7,6,5,4,3) then a new list has been formed by *consing* a (9) to the tail (6,5,4,3). During the *consing* of (9) the pointer offset is compared with the last used offset, LastUsed. If it is the same and less than the block size then it is simply incremented, the new entry made and LastUsed updated. This would have occurred as the integers (6), (7), (8) were added. If on the other-hand the pointer offset is less than the LastUsed a *cons* is being applied to the tail of a longer list, as is the case with the (9). In this case a new list block must be allocated and its Base-Offset pointer set to the tail contained in the original list. The offset part being set to the point in tail that must be extended. The new entry can now be made and additional elements added. As would be expected the two lists now share a common tail, just as would have been the case with a Linked List implementation. If the new block becomes filled then, just as before, a larger one by the factor $\frac{1}{r}$, is added and new entries continued. Often, for the cost of a comparison, the new block allocation can be avoided completely. If the next location happens to contain an element identical to the one being *consed*, as is frequently the case in some applications, then the pointer only need be updated, conceptually a simple form of "hash consing".

As can be seen, the majority of the accesses are to adjacent locations yielding a significant improvement in the sequential access time. Locality makes efficient use of cache line loads.

Most functional language implementations come with in-built special functions that allow indexing to the nth element of a list or finding the length of a list. These are normally implemented by costly linear time recursive functions. With the VList structure, random probes to the nth element take constant time on average. Consider starting with a list pointer in Fig 1 then to find the nth element subtract n from the pointer offset. If the result is positive then the element is in the first block of the list at the calculated offset from the base. If the result is negative then

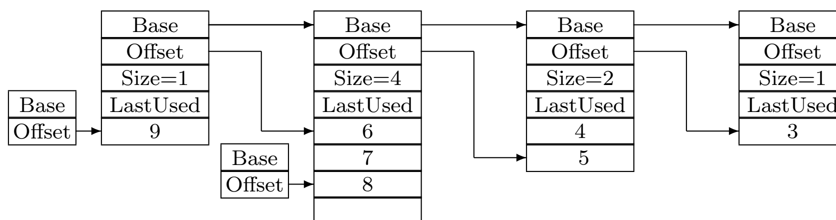

Fig. 2. A Shared Tail List

move to the next block using the Base-Offset pointer. Add the Previous pointer offset to the negative offset. While this remains negative keep moving onto the next block. When it finally becomes positive the position of the required element has been found.

To compute the average access time notice that, for random accesses, the probability that the element is found in the first block is higher than in the second and higher still than in the third in proportions dependant on the block size ratio r chosen. Therefore, the time becomes proportional to the sum of the geometric series.

$$1 + r + r^2 .. \; or \; \frac{1}{1-r}, \; a \; constant$$

To compute the length of a list the list is traversed in the same way but the offsets are summed. Since every block must be traversed this will typically take a time proportional to the number of blocks. If, as is the case in Fig 1, r = 0.5 this would yield O(lg n), a considerable improvement over the O(n) time for a Linked List.

Table 1 give the comparison results for 100,000 operations using a standard linked list and a VList. The operations include *cons* - add element to list, *cdr* - tail of list, *nth* - index to each of n elements and *length* - compute list length.

Table 1. Comparison Linked List and VList(mS)

Operation	**LList**	**VList**
cons	296	21
cdr	23	7
nth	days	38
length	54	0.011

2.2 Refining the VList

The requirement to use two fields, base and offset, to describe a list pointer becomes cumbersome. Firstly there is the time penalty for two memory accesses during storage or retrieval and secondly the additional space required, twice that of a normal pointer. It would be more efficient if a single pointer could be used to represent the list. However, to achieve this it must be possible to recover the base

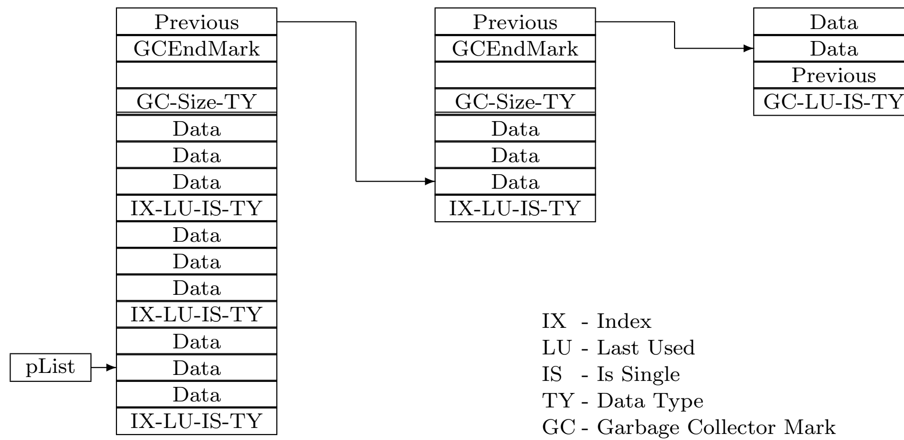

Fig. 3. VList With Single Pointer

of a list block from a simple list data element pointer, given that the data element itself may be an 8 bit character, 16 bit word or 32 bit integer.

This trick can be accomplished by breaking the list block into 16 byte sub-blocks, each one double word aligned in memory. The last 4 bytes in the sub-block are reserved for an index that is the offset of the sub-block from the block base and a data type field. The other 12 bytes are available for data. See Fig 3. Thus with this arrangement the sub-block base is found by masking out the lower 4 bits of the pointer, the block base then being simply calculated from the sub-block index.

To enable small lists to be represented efficiently the first block allocated to a new list is structured differently. It contains only two locations for data, the other being reserved for the Previous pointer. The flag IsSingle allows this type of block to be differentiated from the multiple entry one. With this arrangement the degenerate VList becomes a Linked List with the same overhead. It now becomes apparent why a 16 byte sub-block size has been chosen rather than for example a 32 or 64 byte one. The larger size would reduce the average overhead for large lists but have the consequence of a very high overhead on small lists.

Several benefits accrue from this list representation. The double word alignment implies that on most modern processors the whole sub-block will be fetched in one cache line. It is therefore cheap to find the data type and thus the data type size for incrementing in the *cdr* or *cons* operations. For the same reason sequential access to the adjacent data elements is rapid too.

Along with the index, the reserved 4 bytes contains the last used element offset in a sub-block (4 bits), the data type (4 bits) and the type of sub-block, single or multiple (1 bit) leaving 23 bits for the sub-block index. The remaining 12 bytes are available in each sub-block for storing data. The reserved part represents a constant 33% overhead for data stored. With this arrangement each sub-block can contain twelve 8 bit characters, six 16 bit words or three 32 bit data types (list pointer, integer, float, etc.) A further overhead is incurred as the last block is, on

average, half filled. The amount depends on the value of the growth ratio chosen, for $r = 0.5$ this would be another 33%.

2.3 Extending a VList

Assume that a pointer references the head of a character VList and a new character element must be added. The sub-block base is recovered from the pointer by masking out the lower 4 bits. The data type size is recovered from the Element Type field, in this case a one byte character. The offset from the base to the pointer is calculated if this is less than 11 then the list pointer is incremented by one and the new element stored at the referenced location. The LastUsed field is updated and the new head of list pointer returned.

If the offset was equal to or greater than 11 the memory block size must be recovered from the block descriptor and compared with the current sub-block index. If there are more sub-blocks available then the list pointer is set to the beginning of the next sub-block and the new element added. If no more sub-blocks are available then a new memory block must be allocated and the element added at the beginning.

In static typed functional languages the lists are usually homogeneous while in dynamic typed languages such as LISP the lists are allowed to be hetrogeneous. Type mixing in one list is achieved by creating a new block whenever there is a change of type. This leads to a worst case degeneration to a simple Linked List if there is a type change on every extension of the list.

When compared with Linked Lists the VLists offer considerable space saving. Assuming for the moment that $r = 0.5$, each block added is twice the previous one. Then for an 8 bit data type such as character there will be 33% overhead for the sub-block index and, on average, half of the last memory block allocated will be unused, a further 33% overhead for a total overhead of 80%. For static typed languages a Linked List requires a pointer and character value, assuming this takes two 32 bit words it will represent a 700% overhead. For character lists the VList is almost an order of magnitude more space efficient.

There are some occasions when it would be an advantage to extend the tail of a list. One such case is lazy list generation while another is character buffering. Clearly it is a trivial matter to extend a VList with the same geometric pattern from its last element too.

2.4 Garbage Collection

After a data set is no longer reachable by a program then it must be considered garbage and collectable as free memory. This is typically done as a mark, sweep and copy activity. With Linked Lists the GC algorithm must pass through each element of the list first marking it as potentially free, then ascertaining if it is reachable from the program roots and finally adding unreachable ones to the free list. Notice that with VLists for all types except sub-lists only the memory block descriptor need be inspected or marked during each step of this mark/sweep/copy cycle turning an O(N) process into an O(lg N) one, a significant advantage.

The VList as described does require that large blocks must be used which could be troublesome as a list is consumed. Large parts of the unused list structure may not be garbage collected. Also allocating large memory blocks becomes difficult when heap memory becomes fragmented, perhaps requiring costly de-fragmentation

processes to be performed frequently. By truncating lists during garbage collection, that is reducing the list to just the size needed, and by using a copying garbage collector most of these problems could be avoided.

2.5 Visp

At this point the VList looks a promising data structure. In order to test its viability as the basis for a practical functional language implementation an interpreter was created to use a sub-set of a slightly modified Common Lisp and then benchmarked. The main departure from Common Lisp was the allowed use of infix operators. The interpreter details will not be covered, only the key design principles will be outlined.

Lisp programs are scanned by a simple parser and converted into VList structures directly reflecting the source structure. No optimizations are performed. For these benchmark tests the VLists are extended using $r = 0.5$, each block added to a list is twice the previous one in size. C++ functions were written to perform the fundamental list operations of *car*, *cdr*, *cons*, *reverse*, *length*, *nth* and so on. Arithmetic operations, flow control functions such as *if* and *while* and function definition *lambda* and *defun* were added. Finally the higher order function *foldr* written to enable the more common list operations to be meaningfully benchmarked.

The low cost of indexing invited the inclusion of two native infix operators, "[" and "&[" with a small amount of syntactic dressing to allow writing "L [n]" meaning "nth L n" and "L &[n]" meaning return the tail starting at the nth position in the list.

The complete interpreter including a garbage collector was written as a set of C++ classes and a simple IDE provided to allow programs to be developed interactively. The look and feel of the developer interface is similar to that of OCAML.

2.6 BenchMarking

A set of simple benchmarks, table 2, were written in OCAML and the Visp dialect of Lisp, some code examples are listed in Fig 4. The OCAML versions were compiled both with the native and byte optimizing compilers. The Visp programs were run interactively via the IDE. Table 3 contains the benchmark results.

Stack overflow with OCAML list functions limited the test lists to 40,000 elements. The OCAML Windows runtime support provides time measurement to a 10mS resolution. Visp on the other hand will manipulate lists that fit in memory, a filter on a list of 100 million elements executes without error.

Visp uses the Lisp dynamic scope function call so pays the price in function call times and the overhead of dynamic type checking is apparent in the arithmetic performance. The functions createlist, reverse, clone, length, append and foldr are built in functions of Visp. Filter odd is a hybrid using foldr and an interpreted filter function. The create test uses an iterative interpreted function to build the list.

The space used for a 40,000 integer list in OCAML is reported as 524Kb and in VISP as 263Kb. A 40,000 character list in OCAML is reported as 524Kb and in VISP as 66Kb.

Benchmarks are notoriously difficult to interpret. However, the speed and space requirement differences do suggest that the VList should be considered seriously as

```
// filter OCAML
   let isodd n =  (n mod 2) == 1
   List.filter isodd x
// filter VISP
   defun filter(F L)(foldr(lambda(o i)(if(funcall F i)(o :: i)(o)))L NIL)
   defun isodd (x) (x % 2 == 1)
   filter #'isodd x
// Slow fib OCAML
   let rec fibslow n =
   if n < 2 then 1
   else (fibslow (n-1)) + (fibslow(n-2))
// Slow fib VISP
     defun fibslow(n)(if (n < 2)(1)(fibslow (n - 1) + (fibslow (n - 2))))
```

Fig. 4. Benchmark Code Examples

an alternative to Linked Lists in functional languages.

Table 2. Benchmark Tests

The Test	Description
create list	Create a 40000 list, initialize each element to a common value
create	Create a 40000 list and initiate to a unique value
reverse	Reverse a 40000 list
clone	Create an identical 40000 list from an existing one
append	Append one 40000 list to another
length	Compute length of list
filter odd	Return a 20000 list of all odd memebers in 40000 list
slow fib	Calcs fib numbers using an exhaustive search, function call intensive
calc	Evaluates a lengthy arithmetic intensive expression 100000 times

Table 3. Comparsion of OCAML with VISP (mS)

The Test	OCAMLN	OCAMLB	VISP
create list	20	30	1
create	20	30	73
reverse	20	50	11
clone	40	120	11
append	80	90	12
length	10	40	0.017
filter odd	60	170	139
slow fib	80	1110	7740
calc	20	210	640
GC	10	60	0.011

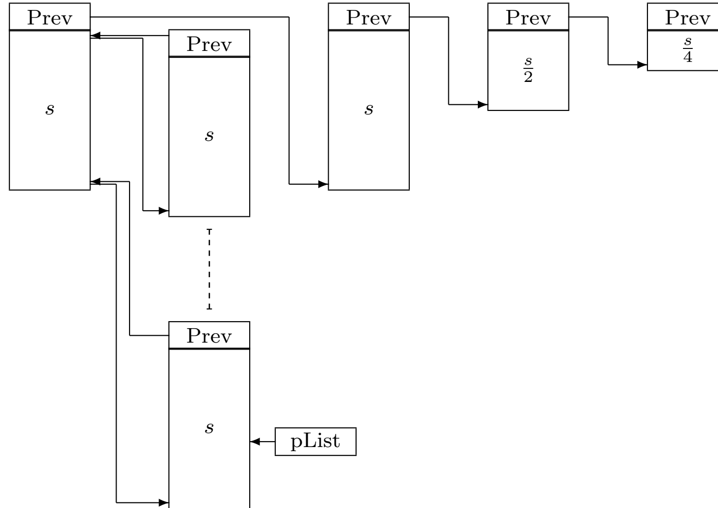

Fig. 5. The n-dimension VList

2.7 The n-dimensional VList

The time constant associated with random access to individual elements in a VList decreases as the growth ratio decreases. However, the size increment grows too and therefore the wasted space. Suppose, as the size grows, that at some point when the block size is s the new block is considered to be a two dimensional array of size s^2. Instead of the new block being arranged as one large block it is arranged as one block of s with pointers to s blocks containing data. This arrangement, depicted in Fig 5, has a negligible impact on *cons* and *cdr* operations while increasing random access times by a small constant time, one extra indirection.

If now as the list grows further the next block required it is considered to be a 3 dimensional array with s^3 entries, and so on. Thus each block is s times larger than the previous one so the random access time becomes

$$\frac{1}{1-\frac{1}{s}} + log_s N \ or \ \frac{s}{s-1} + log_s N$$

For a moderately large s this can be considered a constant for any practical memory size. Notice too that the average waste space also tends to a constant, namely $\frac{s}{2}$.

Recall that the Index in a sub-block was allocated 23 bits to allow the lists to become reasonably large. However, with the n-dimensional arrangement the Index could be restricted to only 7 bits allowing $s = 128$ sub-blocks and implying for integer data types that $3 * 2^7$ or 384 elements can be stored while for character data types this becomes $12 * 2^7$ or 1536. But there are now two more free bytes in a

sub-block that can be used for data so the value can become $14 * 2^7$ or 1792. With this structure the overhead has been reduced to just two bytes per sub-block or 15% for character types such as strings and the waste space restricted to $\frac{s}{2}$. Note 1792^3 is over 4Gb so in a 32 bit environment n need never be greater than 3.

Clearly the garbage collection and space allocation is greatly helped by this arrangement too. Blocks are a uniform size, and never greater than s reducing the problems of memory fragmentation. The constant time for garbage collection however has been relaxed. However, since most allocations are in 128 sub-block chunks garbage collection will still run a respectable $d * 2^7$ faster than a Linked List, where d is data elements per sub-block.

2.8 Thread Safety

Functional programming naturally leads to concurrent processing and therefore the ability to multi-thread the *cons* list extension with VLists is essential. As was seen earlier extensions of a list are controlled by the state of LastUsed in the sub-block. In a multi-thread environment a race condition can occur. Two threads can both inspect the LastUsed and both decide that the next *cons* can be a simple insert at the next location. Perhaps thread one started this activity and context switched to thread two just after the thread one LastUsed test was completed.

VLists can be made thread safe by adding a Thread Lock (TL) bit in the sub-block and using an atomic set-bit-and-test instruction to test its state. TL is normally zero. When a thread wants to update LastUsed it first sets and tests the TL bit.

If the TL bit was zero then the LastUsed is compared with the list pointer being extended. If a new block would be needed then the TL bit is cleared and that activity started. Else the new element is inserted in the next location, the LastUsed updated and then TL cleared.

If the TL bit was set then the thread assumes conflict, adds a new block and points it back to the appropriate list tail in the sub-block being extended.

With this arrangement the list is thread safe but will very occasionally have a less than optimal structure.

3. FUNCTIONAL HASH ASSOCIATIONS

The problem forming associations is endemic to all programming languages. A modified VList structure can provide the basis for a functional hash list with insert and lookup times that approach those of a standard chained hash table with hash table doubling. In Fig 6 the basic form of the VList has been modified so that each block contains two portions, the data and a hash table, to form a hash-list. Thus each time the hash-list grows both the data area and the hash-table grow by the same factor. Each data element is a pair including a list pointer to the entry, normally a list containing the associated items, and a hash-table chain link pointer offset. The hash-table portion is a set of offsets from the block base to the first in a chain of associated items. The link is hidden, a hash-list behaves as a standard homogeneous list of lists allowing all the standard list operations *cons*, *cdr*, etc.

Assuming a suitable hash function, each time a new element is *cons'ed* to the hash list the hash value for the key, normally the first in the association pair, is computed. The offset value located at the corresponding hash table entry is stored in the new hash list entry and this entry offset stored in the hash table entry thus

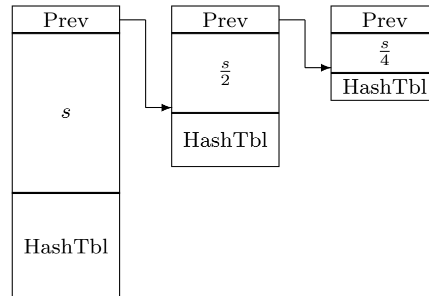

Fig. 6. Hash Table VList

creating a standard chained hash table for all the entries in this memory block.

The search for an association is a little more complicated but can be accomplished in constant time, except in the case of degenerate list formation. Given a hash list, which may be an arbitrary tail of a hash list, then an association is found by first hashing the key. This hash value is used to index the block's hash table and the chain followed until a match is found, using a provided comparison function. If this lies in the tail specified, that is the pointer is less than or equal to the list pointer, then return the entry tail list. If no match is found or the hash table entry was zero, then follow the block previous pointer and repeat for the next block in the list. However, the key hash value does not need to be recomputed. If the end of the list is reached then return the empty list.

As with the VList the probability of finding the association in the first block is higher than in the next block and so on. Hence if the growth is geometric then lookup will be, on average, a constant times the basic hash-table lookup time. This in turn will be governed by the hash table size, number of data entries and lookup time for chained hash tables, a well document relationship. Knuth [1998] and Sedgewick [1998]

Notice the structure is fully functional and persistent though there is a time penalty for degenerate hash lists, e.g. when every tail is extended by one. Also that the performance can be optimized further by allocating the majority of the block to the hash table and reducing it as the actual entries grow. Duplicate entries are found by taking the tail of the returned list and repeating the search on the tail, the order is maintained in the chaining sequence. The structure can be considered garbage collection friendly. All the chain links are offsets contained within a block and can be safely ignored by the garbage collector.

Naturally, the VList concept can be used to create non-functional hash tables too. It is common practice to speed up hash table implementations by periodically doubling the hash table size as the load factor increases. Since each entry must be re-hashed and copied to the new table there is a high price to pay. The alternative is to avoid this by growing the hash table as described above. Shrinking also becomes more efficient. When the total load factor is such that the largest hash table block

is no longer required table entries can be copied to the lower blocks. No re-hashing is required and the operation can be integrated with deletes to become incremental.

4. THE VARRAY

Resizable Arrays are valuable in a wide range of applications and previous work in this area has described how string manipulation, buffers, stacks, queues randomized queues, priority queues and deques can benefit. Brodnik, Carlsson, Demaine, Munro, and Sedgewick [1999] Both RAOTS and the Hash Array Tree(HAT)Sitarski [1996] solution require copying all or part of the data structure while resizing. With the RAOTS space overhead and unused space proved to be proportional to $\sqrt{N}$ while resizing may be accomplished in an $O(1)$ time per addition or deletion. It will be demonstrated with a VArray the cost constants can be improved and copying avoided altogether.

The VArray is based on the VList and utilizes the same principles. Just one of the infinite possible VArray structures can be found in Fig 7. By choosing $r = 0.25$ and an equal sided 2-dimensional sub-block, gives small time constants and a space waste proportional to $\sqrt{N}$. Varying r, the number of dimensions or the side ratio gives VArrays with different time constants or space utilization.

Suppose that the structure is started with $s = 8$. Initially two blocks will be allocated one with eight pointers, the first of which points to the sub-block of eight data locations. Data entries are added until a second sub-block is created and the second entry in the pointer block created. This continues until all the pointer entries are filled. Then the structure is repeated with $2s$ size blocks. The array header is updated to point to this new block and it is updated to point to the previous now complete block. Notice that the append operation is $O(1)$ and no copying takes place as the array grows.

Random accesses can be made to the array and as shown previously will take on average $O(1)$ time with a worst case time of $O(lgN)$. The worst case wasted space occurs when a new pointer block is required. If the previous size was s then the before any entries are made a space of $2s + 2s$ or $4s$ The size already allocated will be $(1 + \frac{1}{4} + \frac{1}{16} \ldots)s^2$ which sums to $1.33s^2$ so the increment will be $\frac{4s}{1.33s^2}$. Thus with the example arrangement for a VArray the space grows proportionally to $\sqrt{N}$. As explained above this can be reduced to a constant by allowing n-dimensional blocks.

The three approaches, HAT, RAOTS and VArray were benchmarked and the comparisons are reported in table 1. Results are included for four versions of the RAOTS, one for each of four different MSB calculations.

The RAOTS algorithm performance is critically dependent on the Most Significant Bit(MSB) index computation. Most modern computer architectures include a leading/trailing zero count instruction for the computation of MSB and LSB indexes. However in situations where these instructions are not available or for portability reasons are exclude Brodnik [1993] provides an efficient alternative. The overall performance of the RAOTS can be improved by optimizing the MSB calculation. A set of solutions, some faster, was found for both the MSB and LSB problem. These are described briefly in section 8.

Notice too that for applications where arrays have periodic size variations VArrays have no risk of the degenerate resize copying which may appear in other solu-

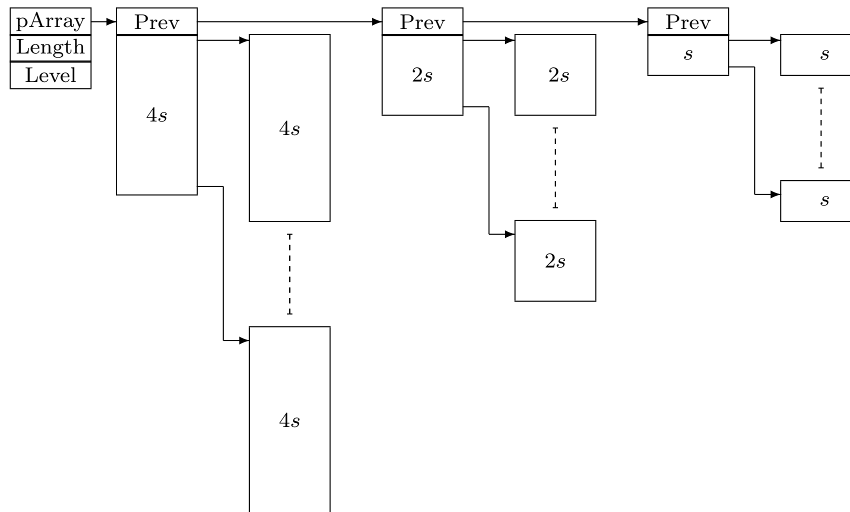

Fig. 7. A VArray ($r = 0.25$)

tions.

4.1 Performance Benchmarks

Each algorithm's performance was measured on an Intel P2, 400 MHz with 512Mb of memory, MS Win NT4 and MS Visual C++ V6.0. In table 4 times are shown in seconds for each benchmark run. The benchmark included 1 million appends to an array, a sequential probe to each of the 1 million elements and random order probe to each element. The four table entries for the RAOTS are for the same test but using one of the four different MSB computations.

Table 4. Resizeable Array Comparison(Sec)

Array	**Append**	**Random**	**Sequential**
VArray	0.096	0.138	0.067
HAT	0.339	0.116	0.036
RAOTS-MSB1	0.120	0.259	0.091
RAOTS-MSB2	0.121	0.244	0.090
RAOTS-MSB3	0.119	0.261	0.134
RAOTS-MSB4	0.121	0.289	0.150

5. DEQUE WITH OR WITHOUT PERSISTENCE

The VList forms an excellent basis both for functional and non-functional deques. Start with the structure in Fig 3, double link the blocks by adding a Next pointer to accompany Prev and form a VDList. Then to access the deque two references must be maintained, one to the front of queue in VDList and one to the back of the

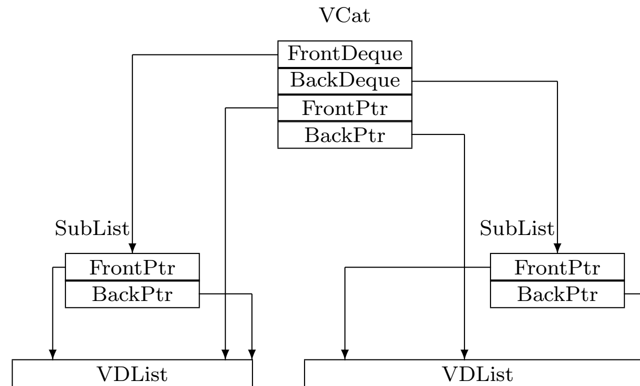

Fig. 8. Persistent Deques

VDList. The pair of references define a sub-list of VDList and thus, as a structure, will be referred to as SubList and is illustrated in Fig 8.

Elements are added to the front of deque by *consing* to the front list contained in SubList and returning a new SubList with the updated list reference. Retrieving an item becomes a *car* at the list position given by back pointer and moving to the next item at the back an update on the back pointer and returning this new SubList structure. When a sub-block or block boundary is passed it becomes available for collection. This process proceeds nicely until two inserts are performed on the same VDList structure. The *cons* works correctly forming a branch but there is no path for a later retrieval to follow this branch. To make this viable, the new branch SubList needs to be treated as if it were catenated to the front of the existing one, an operation to be covered shortly.

The new block size for VDList must be computed differently to that of a Vlist, it cannot simply become a factor $\frac{1}{r}$ larger than the previous one. This would cause the space to always grow even if the retrieval rate is equal to or less than the insert rate. Instead the new block is created with a size that is a fraction f of the actual used space between the front and back of the VDList. With this arrangement the blocks will grow larger as the used space grows, when inserts are faster than retrievals and grow smaller when retrievals are faster than inserts. When the insert and retrieval rates are constant there will be about $\frac{1}{f}$ blocks containing the queued elements. Hence any individual element in the VDList can be randomly accessed in constant time from either end of the VDList. Like VLists, insert and retrieval are also constant time operations. Note using the VArray structure yields non-functional deques that have high space efficiency.

Ideally, functional deques should offer persistence, parts or all of old deques should still be accessible when new ones are derived from them. Kaplan and Tarjan [1995] and Okasaki [1997] show how to do this within a functional programming environment and obtain deques that may be catenated in constant time too. Some of these concepts have been embodied in the development of VDLists to provide similar functionality.

5.1 Catenation

Catenation is appending of one list or SubList to another. The routine, but slow, approach is to copy the appended list. However, it can be accomplished by creating a new data structure, called here VCat, comprises two pointers to elements in a VDList blocks, front and back, along with two pointers to the SubList data structures front and back. Now the branch can be represented by a VCat structure linking the new front SubList and the existing SubList. See Fig 8. The VCat structure can support all the deque operations that the base deque type can support. For example a *cons* can be performed on a VCat of two deques. The the front SubList pointer is followed, if the front element pointer of the VCat is equal to or greater than the one of the SubList then a *cons* is performed on the VDList. A new SubList structure, the two pointers, is allocated reflecting this extension and a VCat structure allocated pointing to this new SubList and with an updated front pointer.

All of the operations can be carried out in this way thus the VCat achieves persistent. With a slight change to the implementation, notice that the same VCat and SubList structures can be used with standard VLists too. Thus with this data structure not only constant time catenation can be achieved both lists and deques but sub-list selection too. However, the *cons* and retrieve operations become proportional to the depth of catenation. At the program level the VCat and SubList structure can be hidden appearing as a standard list or deque. The base list type, VList or VDList, defining what operations are allowed. Only special functions that work directly on the internal structure need be aware of their existence.

6. OPTIMIZING FOR STRING DATA TYPES

The VArray can be adapted to provide the same characteristics as the abstract data type for strings found in many libraries. However in most applications strings tend to be small and a low overhead becomes an important space determinant. With two modifications VArrays provide an excellent storage structure for strings.

First, the VArray is most efficient using two control variables, Size and Level along with a pointer to the root tree node. Strings on the other hand can be handled efficiently with a pointer and string Length only, given that resizing is managed by copy doubling or constant extension. However, given the length of a string in a VArray the corresponding size and level can be derived. Using integer arithmetic,

$$Level = (MSB(Length)/t) * t$$

and

$$Size = shl(shr(Length, Level) + 1), Level)$$

Both these calculations take a small constant time and for the majority of string manipulations are insignificant or for many not even required since length suffices. Thus VArrays for strings can be implemented with just a pointer and Length as the minimum constant overhead.

Second, performance and space utilization may be improved by using the copy resizing for small string sizes. An extra indirection is avoided and the space may be set to closely match the actual string length. This optimization also allows a

C++ class to return a pointer to the actual string. If requested the array structure can be copied to contiguous memory space and the pointer returned. Internally, for example, setting the string length negative could signal this state.

7. CONCLUSIONS

The VList, VDList, VArray, deque and hash-list look like promising solutions to some traditional functional programming problems, giving significant speed gain, space saving and persistence. As described above, some of the data structures have been implemented and benchmarked within the experimental Visp interpreter environment or been coded and compared with previous solutions. An outstanding task is the complete verification of the deques and hash-lists in the Visp environment and benchmarking against alternative solutions both functional and non-functional. This will be the subject of further research.

Such a fundamental change in the basic data structure for lists requires solid characterization with a broad range of actual application programs. There may be some as yet unrecognized problems or surprising benefits. More research is needed.

8. MSB AND LSB INDEX COMPUTATION

The problem of computing the MSB or LSB index provides an interesting challenge yet satisfactory solutions to these apparently simple problems are elusive. Brodnik [1993] developed a compact solution derived from Fredman and Willard computation of $log_2 x$. This computes the LSB in only 29 instructions with no branches but does use multiplication. In their implementation of MSB computation to support RAOTSs Brodnik et al use a constant time algorithm that solves the problem nicely in about $k\ lg\ m$ number of steps using only Add, And, Or and conditional branching. The listing for an implementation of this solution, labeled MSB1, and the others to be described shortly, can be found in Fig 9, while LSB algorithms are in Fig 10. In all cases j represents the input word, where $j > 0$.

The performance of this basic RAOTS algorithm can be improved significantly by optimizing the MSB index computation, as is demonstrated by the comparative benchmark results for RAOTS using MSB1 and RAOTS using MSB2. In general removing conditional branches from an algorithm reduces pipeline stalls and reducing constant sizes thus reducing code length as well as speeding load times. With these principles in mind MSB1 has been reorganized to give the faster functionally equivalent MSB2 with fewer steps.

MSB3 provides a solution that uses no conditional branches. Here the branches have been replaced by using a property of twos complement arithmetic, namely that zero minus one results in all the bits being set, and that the index is given by m, the word length, minus the number of leading zeroes. The first step clears the top bit by shifting j right one position. Then just as in MSB2 the top 16 bits of this shifted word are extracted and one subtracted. If all these bits were zero then all the bits in the result, including the leftmost bit, will be set to one. If set, the left most bit signifies that 16 bit positions were zero so it is shifted right to become the appropriate bit in the result index. j is then shifted left by this amount allowing the next 8 bits to processed. This continues until the index has been computed. Benchmarking shows that this solution is slightly slower than MSB1 and MSB2.

```
 //MSB1 Brodnik, Carlsson, Demaine, Munro & Sedgewick
  r=0;
  if ((new_j = j & 0xFFFF0000)) { r |= 16; j = new_j; }
  if ((new_j = j & 0xFF00FF00)) { r |=  8; j = new_j; }
  if ((new_j = j & 0xF0F0F0F0)) { r |=  4; j = new_j; }
  if ((new_j = j & 0xCCCCCCCC)) { r |=  2; j = new_j; }
  if (         j & 0xAAAAAAAA)    r |=  1;
  return r;

// MSB2
  r=0;
  if (j & 0xFFFF0000){ r |= 16; j>>=16; }
  if (j & 0xFF00){r |=  8;j>>=8; }
  if (j & 0xF0){r |=  4;j>>=4; }
  if (j & 0xC) {r |= 2;j>>=2; }
  return r|((j&2)>>1);

// MSB3
  j>>=1;
  r=(((j&0x7FFF8000)-1)>>27)&0x10;
  j<<=r;
  t=(((j&0x7F800000)-1)>>28)&0x8;
  r|=t;j<<=t;
  t=(((j&0x78000000)-1)>>29)&0x4;
  r|=t;j<<=t;
  t=(((j&0x60000000)-1)>>30)&0x2;
  r|=t;j<<=t;
  return 31-(r|((~(j>>30))&1));

// MSB4
  union{int r;float f;};
  f=(float)j;return((r>>23)+1)&0x1F;
```

Fig. 9. MSB Index Computation

The saving gained by removing branches is offset by the additional steps needed and the constant sizes.

Most modern processors have a built and fast floating point unit which can be exploited to compute the MSB index efficiently. Recall that floating point numbers are represented by a normalized binary mantissa and exponent, the exponent recording the number of positions the mantissa was moved right during normalization and consequentially the index to the MSB. Given this, MSB4 derives the MSB index by simply converting j to a floating point number and extracting the exponent. Surprisingly this is only 30 to 40 percent slower than the other methods, but should be treated with caution when portability is a concern.

The computation of the LSB index commences using an old trick to set all the

```
// LSB1
         j^=(j-1);
         r=(j&0x10000)>>12; j>>=r;
         t=(j&0x100)>>5; j>>=t; r|=t;
         t=(j&0x10)>>2; j>>=t; r|=t;
         t=(j&0x4)>>1; j>>=t; r|=t;
         return r|(j>>1);
// LSB2
         const unsigned int K5=0x55555555,K3=0x33333333;
         const unsigned int KF0=0xF0F0F0F,KFF=0xFF00FF;

    j^=(j-1);
         // Count Population
         j-=((j>>1)&K5);
         j=(j&K3)+((j>>2)&K3);
         j=(j&KF0)+((j>>4)&KF0);
         j+=j>>8;
         return (j+(j>>16))&0x1F
```

Fig. 10. LSB Index Computation

bits below the LSB to one and clear all the bits above it. Then the problem becomes one of counting the number of bits set. In LSB1 a method analogous to MSB3 is used, however, fewer steps are required and the benefits of small constants can be gained resulting in a fast algorithm that uses only 19 Shift, Add, And or Xor instructions.

As an alternative LSB2 uses a tuned version of the Count Population algorithm to achieve an index with 18 instructions in approximately the same time, full scale constants offsetting benefit of fewer instructions.

ACKNOWLEDGMENTS

I would like to thank Prof. Martin Odersky and Christoph Zenger at the Labo. Des Methodes de Programmation (LAMP), EPFL, Switzerland for their review of the draft paper and valuable comments.

REFERENCES

Bobrow, D. G. and Clark, D. W. 1979. Compact encoding of list structures. *ACM Transactions on Programming Languages and Systems 1*, 2 (Oct), 266–286.

Brodnik, A. 1993. Computation of the least significant set bit. In *Electrotechnical and Computer Science Conference, Portoroz, Slovenia,*, Volume B (1993), pp. 7–10.

Brodnik, A., Carlsson, S., Demaine, E. D., Munro, J. I., and Sedgewick, R. 1999. Resizable arrays in optimal time and space. In *Workshop on Algorithms and Data Structures* (1999), pp. 37–48.

Clark, D. 1976. List structure: Measurements, algorithms, and encodings, (ph.d. thesis). Technical report (Aug), Dept. of Computer Science, Carnegie-Mellon University.

Clark, D. W. and Green, C. C. 1977. An empirical study of list structure in lisp. *Communications of the ACM 20*, 2 (Feb), 78–87.

GREENBLATT, R. 1977. Lisp machine progress report. Technical Report memo 444 (Aug), A.I. Lab., M.I.T., Cambridge, Mass.

HANSEN, W. 1969. Compact list representation: definition, garbage collection and system implementation. *Communications of the ACM 12*, 9, 499.

KAPLAN, H. AND TARJAN, R. E. 1995. Persistent lists with catenation via recursive slow-down. In *27th Annual ACM Symposium on Theory of Computing (Preliminary Version), ACM Press* (1995), pp. 93–102.

KNUTH, D. 1998. *The Art of Computer Programming, volume 3: Sorting and Searching, 2nd Ed.* Addison-Wesley, Reading, MA.

LI, K. AND HUDAK, P. 1986. A new list compaction method. *Software - Practice and Experience 16*, 2 (Feb), 145–163.

OKASAKI, C. 1995. Purely functional random-access lists. In *Functional Programming Languages and Computer Architecture* (1995), pp. 86–95.

OKASAKI, C. 1997. Catenable double-ended queues. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)",*, Volume 32 (1997), pp. 66–74.

SEDGEWICK, R. 1998. *Algorithms in C++, 3rd Ed.* Addison-Wesley, Reading, MA.

SHAO, Z. 1994. Compiling standard ML for efficient execution on modern machines (thesis). Technical Report TR-475-94.

SHAO, Z., REPPY, J. H., AND APPEL, A. W. 1994. Unrolling lists. In *Conference record of the 1994 ACM Conference on Lisp and Functional Programming* (1994), pp. 185–191.

SITARSKI, E. 1996. Algorithm alley: Hats: Hashed array trees. *Dr. Dobb's Journal 21*, 11.