# Sequence Types for Functional Languages

Edoardo S. Biagioni
August 1995
CMU-CS-95-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-95-06

**Abstract**

The use of types such as arrays, lazy lists, and other sequential types in Standard ML and other advanced languages can be made as natural and useful as the use of lists. These types are collectively referred to as *sequences*. This report presents a *sequence interface* which can be satisfied by every sequence type regardless of the details of representation, laziness or eagerness, extensibility, and mutability of the specific data structure implementing the type. In addition, an implementation of a sequence type can satisfy more detailed, specific interfaces, some of which are presented. The report also introduces operations for mutating arrays that allow the same style of looping as the conventional list operations and can be implemented efficiently on conventional architectures. The array interface is extended and specialized to allow a efficient implementations of byte arrays.

This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

**Keywords:** Standard ML, signatures, data types, sequence types, sequential access, arrays, byte arrays, data structures, algorithms, compilers, garbage collection

# 1 Introduction

Support for aggregate data types is provided by every non-trivial programming language. These types usually resemble either Fortran arrays or LISP lists, providing either efficient indexing or efficient extension. Modern array-oriented programming languages now provide dynamic allocation of arrays and pointers into arrays (pointer arithmetic), and list-oriented languages provide efficient user-defined recursive data types and pattern matching access to lists. Though many languages, such as C and SML, allow programming with both lists and arrays, the language support for one or the other of these types is usually barely sufficient to allow use of the type. For example, the lack of automatic garbage collection makes lists much harder to use in C than in SML, and the lack of pointers arithmetic makes arrays harder to use in SML than in C.

This report describes types and operations that make it equally easy and useful for SML programmers to use arrays or lazy streams as to use lists. We will refer to these types collectively as *sequences*, since logically a value of any of these types holds an ordered sequence of values. These types and operations have been grouped into a *sequence interface* which can be satisfied by every sequence type regardless of the details of representation, laziness or eagerness, extensibility, and mutability of the specific data structure implementing the type. In addition, each sequence type may satisfy a more detailed, specific interface.

The sequence interface includes many operations traditionally provided on lists; making these operations available for arrays and streams as well as lists makes it possible to write functions that operate on arbitrary sequences, removing the need for distinct but logically identical functions for lists, arrays, streams, and other sequences. In other words, all the benefits of polymorphic generalization accrue by writing programs to operate on sequences rather than specific aggregate types.

The key distinction between sequences and specific types such as lists and arrays is that sequences explicitly allow for a variety of implementations and a variety of asymptotic performances for the same operation. As a result of this definition, the sequence interface must not contain operations specific to a single data structure, and must be designed to be as general as possible. The design of the sequence interface stresses access from the front of the sequence and aggregate access to all elements of a sequence. The former is strictly sequential, the latter does not define access order and therefore permits strictly sequential implementation. The sequence interface can therefore be efficiently implemented by any implementation that allows efficient sequential access, including lists, arrays, streams, and CML (concurrent) channels [22].

Unlike list types in conventional functional languages, array types are mutable. This report describes a set of operations for mutating arrays that allow the same style of looping as the conventional list operations and also can be implemented efficiently on conventional architectures.

Any good interface to arrays must provide for efficient implementation of operations on arrays of bytes; this includes storing the bytes sequentially in memory and access and update using multi-byte words. The *word array* interface is one of the specific sequence interfaces. Word array provides big-endian, little-endian, and native-endian, 1-byte, 2-byte, 4-byte, and 8-byte access to byte arrays. Since efficient byte storage cannot be achieved by a polymorphic, separately-compiled SML program, byte arrays satisfy a *monomorphic* generic sequence interface rather than the more general polymorphic sequence interface.

Section 2 describes in detail the generic sequence interface and a specific sequence interface. Section 3 describes the monomorphic generic sequence interface and a specific interface for byte arrays. Section 4 describes the efficient implementation of specific sequence types, with a focus on efficient implementation of arrays. Section 5 relates the work described here to other work in the field, and Section 6 summarizes the work and draws some conclusions.

# 2 Description

## 2.1 Traditional and Sequence Access to Arrays

The operations used to access traditional lists in LISP are `car` and cdr, which given a list return the first element (head) of the list and what is left of the list after removing the first element (tail). In Standard ML, the corresponding operations are `hd` and `tl`; SML also allows access to lists via pattern matching, which combines a test for the end of a list with access to the head and tail of the list if the list is not empty.

The operation used to access traditional arrays in Standard ML is `sub`, which given an array and an index returns the value of the specified array element. The limitations of this definition include failure to gracefully support sequential access to the elements of an array, which has to be coded as in the following code fragment:

```
fun loop (array, index) =
     if index >= Array.length array then ...
     else ... sub (array, index)
          ... loop (array, index + 1) ...
```

This code fragment shows that array access as currently defined requires integer arithmetic and integer comparison. This style of coding is obscure and error prone, since it is easy to mistakenly use `>` instead of `>=` in the code and since it requires using multiple variables correctly. We compare the above with the equivalent code fragment to sequentially access a list using pattern matching:

```
fun loop nil = ...
  | loop (head :: tail) =
      ... head ... loop tail ...
```

In a loop over the elements of a list, each access tests for the end of the list and, *if the list is not empty*, makes available the first element of the list and the rest of the list. Not only is there no need for integer arithmetic and no possibility of accessing beyond the end of the list, but the convenience, expressiveness, and efficiency of such code makes its use pervasive in SML. For example, the majority of functions in the SML/NJ list library use this pattern to loop over lists.

For a sequence satisfying the signature given in Figure 1 (the complete definition is in Appendix A), the equivalent of this pattern matching access to the front of the list can be obtained by using the function `next` as follows:

```
fun loop NONE = ...
  | loop (SOME (head, tail)) =
      ... head ...
      ... loop (Sequence.next tail) ...
```

It is worth noting some of the differences between list access and sequence access. In SML, the list constructors "`::`" and "`nil`" are defined within a datatype, with the result that they can be used as patterns in a pattern match; however, this also means that the representation of the type is fixed. To allow pattern matching, `next` returns `NONE` for the empty sequence and

`SOME (first element, rest)` for non-empty sequences. The pattern matching that is implicit for lists requires an explicit function call for sequences[1]. This function call may convert from any internal representation to the form of the result, so unlike lists, the representation of the sequence type is not fixed by the signature. Because of this, sequences of many different types can satisfy the same signature.

When looping over arrays, the above loop is much simpler, clearer, and less error prone than the earlier loop using the `sub` operation.

## 2.2 The Sequence Interface

The essential part of the `SEQUENCE` signature is given in Figure 1.

```
signature SEQUENCE =
 sig
  type 'a T
  val next: 'a T -> ('a * 'a T) option

  val new: ('b -> ('a * 'b) option)
         -> ('b -> 'a T)
  ...
 end
```

Figure 1: The `SEQUENCE` signature

The dynamic semantics of `new` and `next` are simply:

$$
\begin{array}{lll}
\texttt{next (new f arg)} \equiv & & \\
\quad \texttt{f arg} = \texttt{NONE} & \Rightarrow \texttt{NONE} \\
\quad \texttt{f arg} = \texttt{SOME (h, t)} & \Rightarrow \texttt{SOME (h, new f t)}
\end{array}
$$

This semantics explicitly does *not* define the order in which successive calls to `new` take place.

The semantics shows that the only real constraint on a type that satisfies the sequence interface is that the elements be accessible in the order in which they are defined. The complete signature in Appendix A also shows that sequences need not be extensible (only "shrinkable", by using `next`). For some sequences, such as lists, extension is natural; for others, such as arrays, extension is normally not available. The generic interface only provides those operations which can reasonably be expected to be available for all sequence types. The value of a data structure which can only be shrunk and not extended can be seen by the fact that an entire implementation of the Fox Net TCP/IP protocol stack [9] relies on such shrinkable, non-extensible arrays to store incoming and outgoing data; for this application, the efficiency of the array representation is more important than the inability to extend the array.

The `new` operation is defined so that it can support lazy as well as strict, finite as well as (potentially) infinite sequences; for example, the definition requires no integer parameters which might limit or pre-define the length of the sequence. The `new` operation is also specifically designed to be used together with `next` to easily build new sequences from existing sequences.

[1]The explicit function call also limits the de-structuring that is possible with pattern matching, so that for example a single test can de-structure four elements from the front of a list, but not from the front of a sequence.

### 2.3 Generic and Specific Sequences

The `SEQUENCE` signature is a *generic* signature that is satisfied by all implementations of sequences. Since the representation of a sequence is not defined by the signature, the signature can be satisfied by structures whose underlying representation is a lazy stream, an array, a list, or any other structure that supports sequential access. We can compare this generic signature to a specific signature that fixes the representation to be, for instance, that used for lists, shown in Figure 2.

```
signature LIST_SEQUENCE =
 sig
  include FINITE_SEQUENCE

  datatype 'a list = nil
                   | :: of 'a * 'a list
  sharing type T = list
 end
```

Figure 2: The `LIST_SEQUENCE` signature

In fact, we can have a whole hierarchy of signatures for sequences, with each signature being more specific than the signature above it, and more generic than the signature(s) below it. Such a hierarchy is shown in Figure 3.

```
             SEQUENCE
          /     |     \
     FINITE   STREAM CHANNEL
      /   \
 ARRAY   LIST
```

Figure 3: A Hierarchy of Signatures derived from `SEQUENCE`.

Streams and channels are both potentially infinite sequences of elements; a stream computes elements on demand, a channel lets threads produce and consume elements at their own pace. Finite sequences support operations such as `append`, `reverse`, and `length` which are not useful for infinite sequences. Arrays and lists are both finite sequences; lists are immutable and can be extended by adding new elements at the front, arrays are mutable but cannot be extended. Some operations, such as `seek`, are asymptotically less expensive on arrays and other operations such as `cons` (list construction) are asymptotically less expensive on lists.

### 2.4 Mutable Sequences

In analogy with the `sub` operation, SML arrays support an `update` operation. Again, the conventional loop to update all the elements of an array in sequence is error prone and requires integer arithmetic:

```
fun loop (array, index) =
     if index >= Array.length array then ...
     else (update (array, index, value);
            ... loop (array, index + 1))
```

This loop can be rewritten to be more like the sequence access code fragment above, giving:

```
fun loop NONE = ....
  | loop (SOME updatable) =
     loop (Sequence.update (updatable, ...))
```

While the design of the sequence access operations was based loosely on the design of existing primitives for accessing lists, there is no existing standard in functional languages for sequential update[2]. The update operation was designed from scratch using the same principles explained above: no integer arithmetic should be required, the operation should be self-limiting with no possibility of going beyond the end of an array, and sequential access should be natural and if possible be a loop over a single variable. The simplicity of the current definition (shown in Figure 4) fails to illustrate the many pitfalls possible in designing such an interface. For comparison, it may be noted that the loop is less complex and error-prone than comparable loops in array-based languages such as Fortran or C.

```
signature ARRAY_SEQUENCE =
 sig
  include FINITE_SEQUENCE
  type 'a updatable
  val write: 'a T -> 'a updatable option
  val rev_write: 'a T -> 'a updatable option
  val read: 'a updatable -> 'a T
  val update: 'a updatable * 'a
           -> 'a updatable option
 end (* sig *)
```

Figure 4: The `ARRAY` signature

In the array signature, an `updatable` is a new type that is interconvertible with the type `T` of readable sequences. An update to an array has the side effect of storing the value into the array at the head position, and also returns a new updatable array into which the next element can be stored, or `NONE` if the end of the array has been reached.

# 3 Monomorphic Sequences and Word Arrays

## 3.1 Monomorphic Sequences

The sequences described in the previous section are polymorphic, that is, the source code of the implementation of any one of the sequence operations will not make any assumptions about the

[2]The `vector-push` operation in Common Lisp [23] is similar to `update`, but differs in that the return value is not a new updatable array; furthermore, the return value cannot be tested easily to check for the end of the array until *after* an update has failed.

representation of data. If the implementation of a sequence is compiled separately from the code that uses the sequence, as is often the case, this restriction is extended to the compiler. The compiler must then store a pointer to each object rather than the value of each object. For small objects, and particularly individual bytes, this boxing overhead is significant. This overhead can be avoided by making the type of the array elements known to the compiler. These sequences cannot satisfy the polymorphic signature shown in Figure 1, but can satisfy the monomorphic sequence signature shown in Figure 5 (the full definition appears in Appendix B).

```
signature SEQ =
 sig
  type element
  type T
  val next: T -> (element * T) option

  val new: ('b -> (element * 'b) option)
         -> 'b -> T
  ...
 end
```

Figure 5: The SEQ signature for Monomorphic Sequences

This generic signature for monomorphic sequences is specialized, in analogy with the polymorphic signature shown above, for monomorphic lists, streams, and channels, but the monomorphic definition is especially useful for arrays. Arrays have very little storage or access overhead per object and are thus quite efficient in storing objects that are small, such as individual bytes of data. The ARRAY_SEQ signature is the monomorphic version of the ARRAY signature shown in Figure 4.

### 3.2 Big-Endian and Little-Endian Access

When accessing bytes in arrays it is often useful to access them as multi-byte words. This is common in systems code which, given arrays of bytes supposedly containing data in a certain format, has to efficiently *unmarshal* them, i.e., parse them, into data structures accessible to a program, and also *marshal* data structures into arrays of bytes. There are two common ways to store a multi-byte integer into a byte array, referred to as *little-endian* and *big-endian*. On any given conventional architecture, either one is usually much faster than the other, so a module that offers multi-byte access should allow big-endian, little-endian, and native byte access. The signature for such a module is shown in Figure 6.

Since both Big and Little as well as the (native) endian array itself are arrays, both read and write operations are provided for every endianness.

### 3.3 Word Array

Defining one endian array for each size at which multi-byte integer access is supported and providing conversion functions between the different array types gives the WORD_ARRAY signature, shown in Figure 7.

The conversion functions defined by WORD_ARRAY allow graceful handling of two common problems that can occur when accessing a byte array with multi-byte operations: unaligned access and

```
signature ENDIAN_ARRAY =
 sig
  include ARRAY_SEQ
  structure Big: ARRAY_SEQ
  structure Little: ARRAY_SEQ
    sharing type T = Big.T = Little.T
        and type element = Big.element
               = Little.element
  val native_big_endian: bool
 end (* sig *)
```

Figure 6: Signature for Arrays with Big, Little, and Native Endianness

```
signature WORD_ARRAY =
 sig
  type T
  structure W8 : ENDIAN_ARRAY
  structure W16: ENDIAN_ARRAY
  structure W32: ENDIAN_ARRAY
  structure W64: ENDIAN_ARRAY
   sharing type W8.element  = Word8.word
       and type W16.element = Word16.word
       and type W32.element = Word32.word
       and type W64.element = Word64.word
       and type T = W8.T
  exception Unaligned
  datatype source = Array8  of W8.T
                  | Array16 of W16.T
                  | Array32 of W32.T
                  | Array64 of W64.T
  val convert8 : source -> W8.T
  val convert16: source -> (W16.T * W8.T)
  val convert32: source -> (W32.T * W8.T)
  val convert64: source -> (W64.T * W8.T)
 end (* sig *)
```

Figure 7: The WORD_ARRAY signature

odd bytes.

- Unaligned access occurs when a byte array is accessed using an $n$-byte operation at a byte index that is not a multiple of $n$. This is not supported by most modern computer architectures and (as a result) by programming languages on such architectures. Since unaligned access is often a result of programming error, and since all sequence access operations maintain alignment of an aligned array, the conversion functions raise the `Unaligned` exception if a conversion would produce an array whose alignment is not correct for the element type of the target of the conversion.
- An array with $m$ bytes in it, when converted to an array that is accessed using $n$-byte operations, has $m \bmod n$ bytes that are not accessible. These bytes (if any), are returned as part of the conversion operation, and also will return to being part of the array if the array is converted back.

The conversion is designed to maintain all sharing between the source and the target of the conversion, so that any update of the target affects the source and vice-versa.

Besides endian and multi-byte access, the one overriding consideration for byte array access is efficiency of implementation. This concern is addressed in the next section.

# 4 Implementation

## 4.1 List Sequences

For most sequence types, implementation is straightforward. List sequences, for example, have an interface that is similar to that of lists; the implementation can use the same underlying representation and most of the operations with no change. The implementation of lists is shown in Appendix D.

## 4.2 Portable Array Implementation

For arrays, one implementation strategy is to use the built-in `sub` and `update` operations to implement the sequence operation. An array sequence can be represented as a triple of type

```
type 'a T = 'a array * int * int
```

The two integers are the first and the last valid index into the underlying array. Successive accesses or updates from the front will increase the first index, and from the back will decrease the last index; when the last index is less than the first, the array is empty.

This implementation is portable and relatively simple, and since sequence arrays are easier to use and provide greater functionality (e.g., shrinking arrays) than regular arrays, the resulting implementation is useful in itself. This implementation has been built and is used within the Fox Net implementation of the standard TCP/IP protocol stack.

The disadvantage of this portable implementation is its sub-optimal performance. On each access, the first and last pointer are compared to insure the validity of the access, then the access is performed after the implementation verifies that the access is legal by comparing the index to the size of the array, and finally a new triple is allocated in memory to hold the resulting sequence array, which differs in exactly one field from the preceding array. In addition, each access performs a function call and return. Each of these operations has constant, small cost and therefore does not affect the algorithmic complexity of the operation, but in practice the total cost is high compared to the cost of a tight loop in assembly or in C or Fortran.

### 4.3 Optimized Array Implementation

A more efficient implementation uses two pointers to represent an array sequence, one to the first and one to the last element of the array. On each access, the two pointers are compared, the access is performed with no further checks, and one of the two pointers is updated. Since such an implementation cannot be expressed in any safe language, the type itself and at least the operations `new`, `next`, and (for efficiency) `seek` must be part of the language implementation, with the further advantage that the operations can be in-lined and there need be no procedure call overhead. The two pointers can be kept in registers, at least within tight loops, to avoid the overhead of heap-allocating a new record on every iteration. The resulting arrays can be every bit as efficient as arrays in array-oriented languages, while at the same time automatically preserving the safety that is the hallmark of functional languages.

### 4.4 Performance of Array Implementations

To fairly compare the two implementations, we have coded an implementation of both `next` and `update` in C using the two implementation strategies, and compared them to a portable SML implementation. The C code of the two implementations is shown in Appendix F and G. Each tests performs 10 million accesses and divides the total wall-clock time by the number of accesses to obtain a time (reported in nanoseconds) for each access. Each reported time is the fastest of ten consecutive runs on an otherwise unloaded machine.

The result of these tests is reported in Table 1 for a DEC PMAX 5000/200 (MIPS architecture) machine with 72MB of memory, and in Table 2 for a DEC AXP 3000-400 (Alpha architecture) machine with 96MB of memory.

| test | read | update | compiler | code |
|---:|---:|---:|---:|---:|
| SML | 2309 | 3707 | SML/NJ 107 | App E |
| unopt. | 1218 | 1437 | gcc 2.6.3 -O2 | App F |
| optimized | 267 | 501 | gcc 2.6.3 -O2 | App G |

Table 1: Performance (ns/loop) of sequence array implementations on MIPS.

| test | read | update | compiler | code |
|---:|---:|---:|---:|---:|
| SML | 1000 | 1375 | SML/NJ 108.3 | App E |
| unopt. | 747 | 937 | gcc 2.6.3 -O2 | App F |
| optimized | 90 | 86 | gcc 2.6.3 -O2 | App G |

Table 2: Performance (ns/loop) of sequence array implementations on Alpha.

As these tables show,

- The "SML-equivalent" C code shown in Appendix F is at least an approximation of the SML implementation, especially for the Alpha. Some of the performance differences can be explained by assuming non-negligible garbage collection costs for SML and because the SML code (unlike the C code) returns a pointer to a pair of the first value and the new sequence array rather than just the new sequence array.

- The optimized implementation shown in Appendix G demonstrates a very substantial performance benefit over the SML-equivalent implementation.

Since the optimized implementation is equivalent to a highly-optimized C pointer iteration, such an implementation of sequence arrays would provide functional languages with safe arrays that require no performance penalty over unsafe arrays.

### 4.5 Limitations

A general warning about these numbers is that they may not be representative of the performance of word arrays when integrated into a full language implementation. While there is no known reason why the optimized implementation should be any faster (or slower) than an equivalent implementation in a compiler, in practice the test is very simple and small details (e.g., a compiler being unable to reserve two registers to one value, and thus imposing storage allocation in the inner loop) could substantially alter these figures.

Another caveat of the optimized implementation is that it may complicate the garbage collector or other automatic storage management system, since the pointers will not always be to the beginning of arrays and any data header adjacent to the beginning of the array may therefore be harder to obtain. Algorithms for tagless (type-directed) garbage collection have been described in detail elsewhere [2, 24, 1], and may be useful in addressing this issue.

## 5 Background

### 5.1 Programming Languages with Arrays

Fortran arrays are described by John Backus in his history of Fortran [7]. Language support for arrays in Fortran includes convenient iterative constructs, and (unsafe) array subscripting and update using special syntax.

Arrays in C are described by Kernighan and Ritchie in the original C language manual [16]. C arrays have the same operations as Fortran arrays, and in addition provide the ability to maintain a pointer into the middle of an array and arithmetic on such pointers, allowing (unsafe) array access and update without the need for explicit integer arithmetic and subscripting. C also provides dynamic allocation and deallocation of memory, allowing the implementation of lists, trees, and other dynamic data structures.

Modula-3 [19] is another modern language with arrays as the fundamental aggregate data type. Unlike C or Fortran, Modula-3 array accesses are safe, that is, the indices are always checked against the bounds of the array before the access is performed. As in the case of SML arrays, this safety comes at a cost in performance, and Modula-3 lets programmers turn off bounds checks when access performance is crucial.

### 5.2 LISP and Scheme

The properties of LISP lists are described by John McCarty in his history of LISP [17]; modern versions of LISP include Common Lisp [23] and Scheme [21]. Language support for lists in LISP includes accessing the first element of a list and obtaining a list with all but the first element, and also modifying the first element of a list and the spine of a list. Garbage collection has also been a part of the language from the beginning, allowing the safe implementation of dynamic data structures such as trees and graphs. Common Lisp and Scheme both support arrays (called *vectors* in Scheme) by providing access and update operations and looping constructs.

Common Lisp, in particular, has a `sequence` type, described in Chapter 14 of the Common Lisp book [23]:

> The type `sequence` encompasses both lists and vectors (one-dimensional arrays). While these are different data structures with different structural properties leading to different algorithmic uses, they do have a common property: each contains an ordered set of elements.

As this quote shows, the motivation for Common Lisp sequences is the same as the motivation for the sequence interface described in this report is the same. One substantial difference between the two is that Common Lisp provides mostly aggregate operations for sequences, whereas this interface provides, in addition to common aggregate operations, element-by-element operations that can be used within loops to efficiently implement all the aggregate operations provided by Common Lisp. Since it is not immediately obvious how a compiler might optimize the `elt` function to avoid bounds checks, the abundance of aggregate operations, each of which can amortize bounds checking over all the iterations of a loop, might plausibly be attributed at least in part to the resulting difficulty of efficiently implementing the aggregate operations if they were defined in user code. Likewise, the Obviously Synchronizable Series Expressions developed by Waters [27, 25, 26]) provide aggregate operations only, with the explicit goal of optimizing loops. Obviously Synchronizable Series Expressions go beyond the work presented in this report in that storage for intermediate results is automatically eliminated where possible. No such result is claimed for sequence types.

Another substantial difference between the sequence interface presented here and Common Lisp sequences is that Common Lisp sequences are specifically restricted to be implemented as either vectors or lists, whereas the sequence interface described in this report is specifically designed to be implemented by a variety of different data structures, including user-defined data structures.

### 5.3 Standard ML and Haskell

The Standard ML language is defined formally by Milner, Tofte, and Harper [18]. The language is distinct from LISP in providing static types and a rich module system, including interface inheritance. Lists in SML resemble lists in LISP, with the distinction that SML provides pattern matching to support list decomposition, that built-in SML lists are immutable, and that all elements of an SML list must have the same type. Arrays in SML are not part of the standard, but where implemented, such as by SML of New Jersey [3] (SML/NJ), share the same interface as LISP arrays, again with the distinction that all elements of an SML array must have the same type.

Arrays in SML are generally polymorphic. In recognition of the greater efficiency of storage possible with monomorphic arrays, the current draft for the SML/NJ standard library [5] includes a definition of monomorphic arrays; this is a generalization of earlier versions of the library [4] in which the only monomorphic arrays supported were arrays of bytes and of reals. The Common Lisp function `make-array` likewise allows the specification of a type for elements of an array, presumably with similar benefits.

The Haskell language [15] is a purely functional, lazy (normal-order evaluation) language. Haskell defines lazy lists, which are equivalent to the stream type of an eager languages, and the preliminary definition of standard libraries [12] defines lazy arrays. Array subscripting is conventional, while array update allows updating multiple elements at once and is purely functional, returning a new array with the specified elements changed.

### 5.4 Other Aggregate Types

The FP language [6] provides *vectors* that resemble the sequences described in this report in allowing both array-like indexing and list-like extension and looping. These vectors are unlike sequences in that they are a single type, so that any actually implementation of the language has to choose between list-like performance and array-like performance, i.e. accepting $O(n)$ cost for either the prefixing or the indexing operation, or must employ some other data structure (e.g. the purely functional random-access lists of Okasaki [20]) with at best $O(\log n)$ for indexing[3]. The generic sequence interface presented in this report allows but *does not require* that a sequence be capable of extension, and this makes it possible to use it as an interface to both lists and arrays.

Larch [14] is a formal specification system for programs. The sequence specification given by Moormann Zaremski and Wing [28] was one of the inspirations for the development of the sequence interface described in this report, though the inspiration was somewhat indirect since the sequence trait described by Moormann Zaremski and Wing differs substantially from the sequence interface in this report.

### 5.5 Motivation: The Fox Project

The original motivation for sequences came from the Fox Project [13]. The Fox Project develops systems software in SML in order to improve both the practice of systems programming and advanced language support for systems programs. Specifically, the Fox Project has built an implementation of the standard TCP/IP networking protocol stack [9, 8] entirely in SML[4]. One of the tasks of a protocol stack is to strip headers from incoming packets, which are represented as byte arrays. The notion of "shrinkable" arrays was developed as an elegant solution to this issue, and was eventually generalized to the sequences described in this report. Word arrays were developed as more elegant and convenient (and potentially faster) replacement for `sub` and `update`.

It should be noted that SML/NJ provides primitives for unsafe access to byte arrays. Experience [11] with this "feature" of the language was part of the original motivation for the development of *safe* high-performance array access interfaces.

## 6 Summary and Contributions

The concept of sequences is not new; what this report describes that is new is the use of sequences to express the common features of many different data structures in established programming languages. The provision of different, incompatible interfaces for different aggregate types is mostly a historical accident, and this work is one step towards rectifying it.

In addition to presenting common interfaces for different types, this report presents a new implementation for arrays that provides the safety required by advanced languages and is able to support substantially improved performance over traditional implementations. The resulting performance is similar to that of implementations of arrays in unsafe languages. This performance benefit is coupled with an improved interface that, compared to commonly available operations for array access, lets programmers use arrays more naturally and reduces the likelyhood of coding errors.

The word array interface extends the possibility of high-performance implementation to arrays of bytes, supporting safe access at different word sizes to allow the highest possible performance.

[3] In previous work the author produced an implementation [10] of the FP language that uses arrays to implement sequences, accepting the corresponding high cost of prefixing.

[4] This implementation runs in user space and is therefore layered above an operating system and its device driver.

Finally, this report describes a new programming interface for modifying the contents of arrays. In the same manner as the interface for reading arrays, the interface for writing arrays also reduces the likelyhood of programming errors and is suitable of high performance implementation.

## References


[1] S. Aditya, C. H. Flood, and J. E. Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proc. 1994 Lisp and Functional Programming*, Orlando, Florida, June 1994.

[2] Andrew A. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2, 1989.

[3] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third International Symposium on Programming Languages Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[4] AT&T Bell Laboratories. *Standard ML of New Jersey – Base Environment*, February 15 1993.

[5] *A New Initial Basis for Standard ML*, April 12 1995. unpublished.

[6] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 16(8):613–641, August 1978. Turing Award Lecture.

[7] John Backus. The history of FORTRAN I, II, and III. In Richard L. Wexelblat, editor, *History of Programming Languages. Proc. ACM SIGPLAN conference*, pages 25–74. Academic Press, June 1978.

[8] Edoardo Biagioni. A structured TCP in Standard ML. In *Proc. SIGComm '94*, pages 36–45, London, England, August 1994.

[9] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian Milnes. Signatures for a network protocol stack – a systems application of Standard ML. In *Proc. 1994 Lisp and Functional Programming*, Orlando, Florida, June 1994.

[10] Edoardo S. Biagioni. Fpc: A translator for fp. Technical Report TR88-027, Dept. of Computer Science, UNC-Chapel Hill, Chapel Hill, North Carolina, 27599-3175, May 1988.

[11] Edoardo S. Biagioni. Program verification for optimized byte copy. Technical Report CMU-CS-94-172, School of Computer Science, Carnegie Mellon University, July 1994.

[12] K. Hammond, J. Peterson, L. Augustsson, B. Boutel, W. Burton, J. Fairbairn, J. Fasel, A. Gordon, M. M. Guzmán, J. Hughes, P. Hudak, T. Johnsson, M. Jones, D. Kieburtz, R. Nikhil, W. Partain, J. Mattson, S. Loosemore, S. Peyton-Jones, A. Reid, and P. Wadler. *Standard Libraries for the Programming Language Haskell.* ftp://haskell.systemsz.cs.yale.edu/-pub/haskell/report/draft-libraries-1.3.dvi, version 1.3 edition, June 1995.

[13] Robert Harper and Peter Lee. Advanced languages for systems software: The fox project in 1994. Technical Report CMU-CS-94-104, School of Computer Science, Carnegie Mellon University, January 1994.

[14] J. Horning and J. Guttag. *LARCH: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[15] P. Hudak, S. Peyton-Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. *Report on the Programming Language Haskell*. ftp://haskell.systemsz.cs.yale.edu/pub/haskell/report/-report-1.2.dvi.Z, version 1.2 edition, March 1992.

[16] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[17] John McCarthy. The history of LISP. In Richard L. Wexelblat, editor, *History of Programming Languages. Proc. ACM SIGPLAN conference*, pages 173–197. Academic Press, June 1978.

[18] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[19] Greg Nelson. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[20] Chris Okasaki. Purely functional random-access lists. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 86–95, 1995.

[21] J. Rees, W. Clinger, H. Abelson, IV N. I. Adams, D. H. Bartley, H. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T Haynes, E. Kohlbeker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, and M. Wand. Revised report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12), December 1986.

[22] John H. Reppy. CML: A higer-order concurrent language. In *Programming Language Design Implementation*, pages 294–305, June 1991. also appears in SIGPLAN Notices, 26(6).

[23] Guy L. Steele Jr. *Common LISP: The Language (Second Edition)*. Digital Press, 1990.

[24] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 Lisp and Functional Programming*, Orlando, Florida, June 1994.

[25] Richard C. Waters. Obviously synchronizable series expressions: Part I: User's manual for the OSS macro package. Technical Report AI Memo 958a, MIT A.I. Lab, March 1988.

[26] Richard C. Waters. Obviously synchronizable series expressions: Part II: Overview of the theory and implementation. Technical Report AI Memo 958b, MIT A.I. Lab, March 1988.

[27] Richard C. Waters. Automatic transformation of series expressions into loops. *Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

[28] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. Technical Report CMU-CS-95-127, School of Computer Science, Carnegie Mellon University, March 1995.

# 7 Appendices

## A Sequence Signature

```
signature SEQUENCE =
 sig
  type 'a T
  val next: 'a T -> ('a * 'a T) option

  val new: ('b -> ('a * 'b) option)
         -> ('b -> 'a T)

  val seek: 'a T * int -> 'a T

  exception Empty
  val head: 'a T -> 'a
  val tail: 'a T -> 'a T
  val nth: 'a T * int -> 'a

  val isempty: 'a T -> bool
  val wait: 'a T -> unit

  val map: ('a -> 'b) -> 'a T -> 'b T
  val app: ('a -> unit) -> 'a T -> unit
  val fold: ('a * 'b -> 'b)
          -> ('b -> ('a T -> 'b))
  val filter: ('a -> 'b option)
            -> ('a T -> 'b T)

 end
```

## B Monomorphic Sequence Signature

```
signature SEQ =
 sig
  type element
  type T

  val new: ('b -> (element * 'b) option)
         -> 'b -> T

  val next: T -> (element * T) option
  val seek: T * int -> T

  exception Empty
  val head: T -> element
  val tail: T -> T
  val nth: T * int -> element
```

```
  val isempty: T -> bool
  val wait: T -> unit

  val map: (element -> element) -> T -> T
  val app: (element -> unit) -> T -> unit
  val fold: (element * 'a -> 'a)
          -> ('a -> (T -> 'a))
  val filter: (element -> element option)
            -> (T -> T)
 end
```

## C Monomorphic Finite Sequence Signature

```
signature FINITE_SEQ =
 sig
  include SEQ

  val create: element * int -> T
  val tabulate: int * (int -> element) -> T

  val reverse: T -> T
  val append: T * T -> T

  val length: T -> int
  val equal: T * T -> bool
  val eq: T * T -> bool
  val less: T * T -> bool

  structure Rev: SEQ
   sharing type Rev.T = T
       and type Rev.element = element
 end
```

# D Implementation of List Sequences

```
functor List_Sequence
    (structure L: LIST): LIST_SEQUENCE =
  structure Shared =
   struct
    local
     signature S =
      sig
       datatype 'a list =
           nil | :: of 'a * 'a list
       exception Empty
      end
     structure S: S = List
    in
     open S
     infix ::
    end
    type 'a T = 'a list
   end

  open Shared

  fun new f seq =
       case f seq of
          NONE => List.nil
        | SOME (element, new_seq) =>
           List.:: (element, new f new_seq)

  fun create (_, 0) = List.nil
    | create (value, count) =
       List.:: (value,
                create (value, count - 1))

  val tabulate = List.tabulate

  fun next nil = NONE
    | next (op :: result) = SOME result

  fun seek (list, count) =
       ((List.drop (list, count))
        handle _ => nil)

  val head = List.hd
  ...
 end
```

# E SML Test Code

```
fun next (T data, first, last) =
     if first > last then NONE
     else SOME (ByteArray.sub (data, first),
                T data = data,
                   first = first + 1,
                   last = last)

fun update (Updatable data, first, last,
            value) =
     (ByteArray.update (data, first, value);
      let val new_first = first + 1
      in if new_first > last then NONE
         else
          SOME (Updatable data = data,
                           first = new_first,
                           last = last)
      end)
  | update (Rev_Updatable data, first, last,
            value) =
    ...

fun time_next array =
     let fun loop NONE = ()
           | loop (SOME (first, rest)) =
              loop (next rest)
         val start_time = now ()
         val _ = loop (next array)
         val end_time = now ()
         val loops = length array
     in print_times ("memory read", loops,
                     start_time, end_time)
     end

fun time_update array =
     let fun loop NONE = ()
           | loop (SOME updatable) =
              loop (update (updatable, 2))
         val start_time = now ()
         val _ = loop (write array)
         val end_time = now ()
         val loops = length array
     in print_times ("memory write", loops,
                     start_time, end_time)
     end
```

## F SML-equivalent C Test Code

```
sequence_array * read (sequence_array * test)
{
  register int tag = test->tag;
  register char * array = test->array;
  register int first = test->first;
  register int last = test->last;
  register sequence_array * update
          = heap_pointer;
  register char result_char;

  if (first <= last)
  {
    if (first < tag) {
      result_char = array [first];
      update->tag = tag;
      update->array = array;
      update->first = first + 1;
      update->last = last;
      heap_pointer = update + 1;
      if (heap_pointer >= heap_end)
        heap_pointer = dummy_heap;
      return update;
    }
  }
  return NULL;
}
```

```
sequence_array * write (sequence_array * test)
{
  register int tag = test->tag;
  register char * array = test->array;
  register int first = test->first;
  register int last = test->last;
  register sequence_array * update
           = heap_pointer;
  if (first <= last)
  {
    if (first < tag) {
      array [first] = 99;
      update->tag = tag;
      update->array = array;
      update->first = first + 1;
      update->last = last;
      heap_pointer = update + 1;
      if (heap_pointer >= heap_end)
        heap_pointer = dummy_heap;
      return update;
    }
  }
  return NULL;
}

...
  gettimeofday (&start, NULL);
  while (test_ptr)
    test_ptr = read (test_ptr);
  gettimeofday (&end, NULL);
...
  gettimeofday (&start, NULL);
  while (test_ptr)
    test_ptr = write (test_ptr);
  gettimeofday (&end, NULL);
```

# G Optimized C Test Code

```
...
  register char * first = test_array;
  register char * last = test_array +
                          MAX_TESTS - 1;
  register char use_result;
...
  gettimeofday (&start, NULL);
  while (first <= last)
    use_result = *first++;
  gettimeofday (&end, NULL);
...
  gettimeofday (&start, NULL);
  while (first <= last)
    *first++ = use_result;
  gettimeofday (&end, NULL);
```