SML/docs/sequences Draft of Chpt 7 of Basis Manual

Programming with Sequences

David MacQueen

May 2, 2000

This tutorial discusses basic techniques for working with sequences using the facilities provided by the Standard ML Basis library.

1 Introduction

A sequence is a *linearly ordered*, *homogeneous* (all elements of the same type) collection of values. Manipulating finite sequences is an extremely common task in programming, and particularly in functional programming (remember that LISP stands for *list* processing). Our purpose here is to review the facilities provided by the SML Basis library for representing and manipulating sequences, emphasizing common patterns for building and manipulating sequences that apply to all sequence types.

The SML Basis library supports programming with sequences through three representations: lists, vectors, and arrays. Lists are inherited from the Lisp tradition and provide sequences that can be built incrementally and are typically processed sequentially. The `list` type constructor,

```
type 'a list
```

is parameterized on the element type, and the associated list operations are polymorphic in the element type. The list type and its operations are packaged in the `List` structure.

For "random access" sequences we have mutable *arrays* from the Algol tradition and their pure variant, *vectors*. Like `list`, the vector and array type constructors are parameterized on their element types:

```
type 'a vector
type 'a array
```

and they are packaged with their respective suites of polymorphic operations in the Vector and Array structures.

The elements of these "polymorphic" vectors and arrays are normally *boxed*, *i.e.* they are treated as being of a uniform size, typically the natural word size of the machine. This is not space efficient when the natural representation of a particular element value is not boxed (e.g. values that fit in a word, or floating point values). To provide more compact representations for these special

cases, there are a number of monomorphic vector and array types packaged in their respective structures, such as `RealVector`, `RealArray`, `Word8Vector`, and `Word8Array`. These specialized structures conform to the `MONO_VECTOR` and `MONO_ARRAY` signatures.

1.1 Commonalities and Differences

Before getting to the common patterns for processing sequences, let's first consider some basic characteristics that the different flavors of sequences have in common and some characteristics that distinguish them.

Here are the common traits of the sequence types:

- We use the `int` type as the index type for referring to elements of a sequence by position.[1]
- We index elements of a sequence starting from 0.

What are the general properties that distinguish the different forms of sequences?

- Lists are represented using a linked data structure while vectors and arrays occupy a contiguous block of memory as illustrated by the list `[1,2,3]` and the corresponding vector in Figure 1.
- Lists and vectors are immutable *values* that cannot be changed once they are constructed, while arrays are mutable: their contents can be changed by the array `update` function.
- Equality on lists and vectors is structural (assuming the element type supports equality), while arrays are stateful objects, like refs, so equality for arrays is object identity (i.e. pointer equality).
- Arrays and vectors are primitive types, while lists, although built-in, are definable as a datatype with data constructors `nil` and `::`.
- Arrays and vectors are allocated all at once, while lists are allocated incrementally, by successively consing elements onto the front of the list.
- Lists are polymorphic, while vectors and arrays are available in both polymorphic and monomorphic versions.

For historical reasons, lists have a privileged place in ML. There are more built-in operations for lists and there is a special square bracket notation for list expressions and patterns. Lists are a more convenient representation for building sequences incrementally and for combining sequences, but they consume more space and sequential access to elements is slow (linear time compared with constant time for vectors and arrays).

[1] Note that indices of elements of a sequence are never negative, so `word` might have been a more natural choice for the index type.

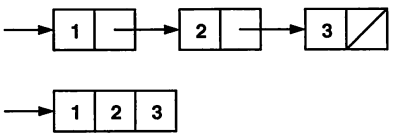

Figure 1: list and vector representations

When the elements of a sequence are all known at the point where the sequence is built, and constant time random access is important, vectors are a good representation. If the size of a sequence is fixed but the elements must change over time, then an array may be the best choice, but using arrays leads to programming with side-effects, which tends to be more error-prone.

An important special case of vectors is the string type, which is essentially the same as a monomorphic character vector. There is a special interface for strings that provides additional functionality that is convenient for string manipulation. We won't cover strings in this tutorial.

2 Common patterns

We'll now turn to the common patterns for working with sequences regardless of the underlying representation. These patterns can be captured by generic operations for creating, accessing, and iterating over sequences.[2]

2.1 Sequence Construction

The primitive patterns for sequence construction vary fundamentally between lists on the one hand and vectors and arrays on the other.

Lists are typically constructed incrementally by "consing" elements on the front, as, for instance, in the recursive definition for append.

Vectors and arrays are constructed all at once. This can be done starting with a list of the elements using `fromList`

`fromList` (*l*: `'a list`) : `'a seq`
: returns a sequence consisting of the elements of *l*

(where in this case `seq` ranges over `array` and `vector`). For building arrays there is also the `array` function that fills the array with a fixed initial element value. [3]

However, there are a couple sequence construction patterns shared by all sequence forms, represented by the functions tabulate and concat.

[2]Some examples will be given below using interactive evaluations on the SML/NJ system. The system's responses are of course unique to SML/NJ and will differ for other compilers.

[3]It is a deficiency of the vector design that there is no direct way of constructing a vector from its elements: one either uses `tabulate`, or one constructs a list of its elements as an intermediate value and then applies `fromList`. SML/NJ provides a special vector expression notation to correct this omission.

`tabulate (n: int) (f: int -> 'a) : 'a seq`
returns a sequence consisting of the elements $f(i)$ for $0 \leq i < n$

`concat (ss: 'a seq list) : 'a seq`
concatenate a elements of the list *ss* into a single sequence

2.2 Accessing elements

Each type of sequence has a function providing random access to the n^{th} element of the sequence: for lists this function is named `nth`, while for arrays and vectors it is called `sub`. For arrays and vectors, the `sub` operation provides constant-time random access to the n^{th} element. For lists, the primitive access method is sequential, with only the first element directly accessible via `hd`, or equivalently, pattern matching on `::`. The `nth` function for lists is defined iteratively, and takes time proportional to the index.

2.3 Attributes

Sequences are finite, so the one basic attribute that we can compute for any sequence is it's length. Thus each sequence type supports a length function.

`length (s: 'a seq) : int`
returns the length of the sequence, i.e the number of (not necessarily distinct) elements in the sequence

A derived attribute is the `empty` predicate, that returns true when applied to an empty sequence. The empty predicate is defined for lists and vector and array slices (described below).

2.4 Iterating over Sequences

The following functions embody the most commonly used sequence iterators. We'll describe their types using a generic `seq` type constructor that can stand for any of `list`, `vector`, or `array`.

`app (f: 'a -> unit) (s: 'a seq) : unit`
apply a function *f* to each element of the sequence *s* for effect, returning the unit value

`map (f: 'a -> 'b) (s: 'a seq) : 'b seq`
return a sequence containing the images of a function *f* on the sequence *s*, in corresponding order

`foldl/foldr (f: 'a * 'b -> 'b) (b: 'b) (s: 'a seq) : 'b`
accumulate the result of a binary combining function *f* operating successively on the elements of a sequence *s* starting with base value *b*

`find (p: 'a -> bool) (s: 'a seq) : 'a option`
return SOME of the first element of a sequence *s* satisfying a predicate function *p* or NONE if none exists

`exists/all (p: 'a -> bool) (s: 'a seq) : bool`
return a boolean value indicating if any/all elements of a sequence *s* satisfy a predicate *p*

`collate (c: 'a * 'a -> order) (s1: 'a seq) (s2: 'a seq) : order`
perform lexicographic comparison of two sequences *s1* and *s2* given a comparison operation *c* on their common element type

The most general and powerful of these iterator functions are the fold operations. There are two variants: `foldl` which accumulates from left to right, and `foldr`, which accumulates from right to left. The fold operations take a binary function and an initial value, and successively combine the elements of the list using the function, and starting with the initial value (illustrated here using the list version).

```
foldl (f,b) [x,y,z] ==> f(z,f(y,f(x,b)))
foldr (f,b) [x,y,z] ==> f(x,f(y,f(z,b)))
```

Note that despite the order in which the elements appear in the terms on the right, `foldl` starts by applying `f` to `x`, then `y`, then `z`, i.e. from left to right. `foldr` applies `f` from right to left. This is the rationale for the directional suffixes "l" and "r".

In both cases, the second argument of `f` is an accumulator argument whose initial value is `b`. The value of `b` and the return value of `f` have to agree in type, but they can have a different type from the list elements. Thus both `foldl` and `foldr` have the type

```
('a * 'b -> 'b) -> 'b -> 'a seq -> 'b
```

The following evaluations illustrate the difference in order of traversal between foldl and foldr.

```
- foldl (op ::) nil [1,2,3];
val it = [3,2,1] : int list
- foldr (op ::) nil [1,2,3];
val it = [1,2,3] : int list
```

Other typical applications of fold are to accumulate the sum or product of a sequence of numbers:

```
- foldl (op +) 0 [2,3,5];
val it = 10 : int
- foldl (op * ) 1 [2,3,5];
val it = 30 : int
```

The power of the fold operations is illustrated by the fact that all the other iterators (except `collate`) are easily (though not necessarily efficiently) defined in terms of folds.

For instance,

```
fun app f l = foldl (fn (x,y) => f x) () l
fun map f l = rev(foldl (fn (x,y) => f x :: y) nil l)
fun find p l = foldl (fn (x,NONE) => if p x then SOME x else NONE
                       | (x,y as SOME _) => y)
                     NONE l
```

As illustrated by the find example, fold operators do not have a short-circuit capability – all elements of the subject sequence have to be processed, even if the final result can be determined by looking at only part of the sequence. Note also that the following simpler definition of map in terms of `foldr` could be used, but only if `f` is pure (i.e. has no side-effects), since mapr applies `f` to the elements of `l` in reverse order.

```
fun mapr f l = foldr (fn (x,y) => f x :: y) nil l
```

2.5 Indexed iterations

For vectors and arrays, any element-wise operations involve selecting elements by their indices, and so it is natural to allow for operations to work not only on the elements themselves but their indices as well. Thus for these sequence types, the iterator functions `app`, `map` (vector only), `fold`, and `find` have variants that operate on index-element pairs (i.e. pairs of the form `(i,sub(s,i))` rather than elements. The index-passing variants have an "i" appended to the function name, e.g. `appi`.

`appi` (*f*: `int * 'a -> unit`) (*s*: `'a seq`) : `unit`
: apply a function *f* to each index-element pair of the sequence *s* for effect, returning the unit value

`mapi` (*f*: `int * 'a -> 'b`) (*s*: `'a seq`) : `'b seq`
: return a sequence containing the images of a function *f* applied to the index-element pairs of the sequence *s*, in corresponding order

`foldli/foldri` (*f*: `int * 'a * 'b -> 'b`) (`b: 'b`) (*s*: `'a seq`) : `'b`
: accumulate the result of a binary combining function *f* operating successively (left-to-right/right-to-left) on the index-element pairs of a sequence *s* starting with base value *b*

`findi` (*p*: `'a -> bool`) (*s*: `'a seq`) : `'a option`
: return SOME of the first index-element pair of a sequence *s* satisfying a predicate function *p* or NONE if none exists

As an example, here `appi` is used to print out a table of the elements of a vector of strings numbered by their indices:

```
fun prvector(v: string vector) =
    Vector.appi
      (fn (i,x) => (print(Int.toString i); print " "; print x))
      v
```

The nonindex-passing form of an iterator can be derived from index-passing version by providing functions that ignore the index parameter. For instance, "`map f s`" is equivalent to "`mapi (f o #2) s`".

2.6 Array Modification

The property that distinguishes arrays is that they can be updated, so when working with arrays it is normal to change their state rather than compute a new array. Thus there is no map function for arrays, and in its place there is an operation `modify` for modifying the elements of the array in place, and a corresponding index-passing version `modifyi`.

`modify (`*f*`: 'a -> 'a) (`*a*` : 'a array) : unit`
replace each element of *a* by its image under *f*

`modifyi (`*f*`: int * 'a -> 'a) (`*a*` : 'a array) : unit`
replace the i^{th} element $x = sub(a,i)$ by *f(i,x)*, for each index `i`

In addition, there are two operations for bulk updating an array: `copy` and `copyVec`. These functions copy the contents of a source array (respectively vector) into a destination array, starting at a specified destination index. These operations raise Subscript if the src array or vector is too big to fit in the destination, or if the destination index is negative.

`copy` *src*`: 'a array,` *dst*`: 'a array,` *di*`: int : unit`
`copyVec` *src*`: 'a vector,` *dst*`: 'a array,` *di*`: int : unit`
copy the elements of *src* into *dst* starting at index *di*

3 Subsequences and slices

One thing one can do with sequences is slice and dice them – that is, extract pieces of the sequence using various selection criteria, and then operate on those subsequences. The treatment of subsequences for lists is quite different from that for vectors and arrays, but there is some overlap of functionality.

For lists, there are filter, partition, and mapPartial. The filter and partition operations select elements of a list sequence using a predicate, returning subsequence(s) consisting of selected elements of the original sequence (`filter f l = #1(partition f l)`). The mapPartial function combines map iteration with filtering based on whether the function returns `NONE` or `SOME`.

`filter (`*p*`: 'a -> bool) (`*s*`: 'a list) : 'a list`
return a subsequence consisting of those elements of the argument sequence *s* that satisfy a predicate *p*

`partition (p: 'a -> bool) (s: 'a list) : 'a list * 'a list`
: return a pair of subsequences consisting of those elements of the argument sequence s that satisfy (respectively don't satisfy) a predicate p

`mapPartial (f: 'a -> 'b option) (s: 'a seq) : 'b seq`
: return a sequence containing the defined (i.e. `SOME`) images of a *partial* (i.e. option-returning) function f on the sequence s

There are also `take` and `drop` operations that return initial and final segments of a list. These can be combined to yield a list analogue of the vector/array slices discussed below.

```
fun slice (l: 'a list) (start: int) (len: int option) : 'a list =
    case len
      of NONE => drop l start
       | SOME n => take (drop l start) n
```

However, this definition of slice for list produces a new list from which one can't recover the original list.

For vectors and arrays, the supported notion of subsequence is a *slice*, which is a contiguous subsequence characterized by starting index and length. For instance, conceptually an array slice is determined by a triple

$$(arr : \texttt{'a array}, start : \texttt{int}, len : \texttt{int})$$

and represents the subsequence

$$(arr[start], \ldots, arr[start + len - 1])$$

where $start$ must satisfy $0 \leq start < length(arr)$ and len must satisfy $0 \leq len < length(arr) - start$. The slice types are abstract, but there is a function `base` that maps a slice back to the representing triple, so it is possible to extract the underlying array.

All of the iterators defined for vector and array sequences (both plain and indexed) have analogs defined for the corresponding slice types, which are packaged in the `ArraySlice` and `VectorSlice` structures.

There are also monomorphic slice structures for each of the monomorphic array and vector structures (e.g. `Word8VectorSlice: MONO_VECTOR_SLICE` for `Word8Vector`). The analog of slice for strings is called a substring.

4 Operating on pairs of lists

It is fairly common to want to process two lists simultaneously, and to support situations there is a ListPair structure that provides analogs of the iterators `app`, `map`, `foldl`, `foldr`, `all`, and `exists` that operate on pairs of lists. There are also functions `zip` and `unzip` for transforming a pair of lists into a list of pairs and vice versa.

When simultaneously iterating over two lists, there is a choice of how to handle the situation where the lists do not have the same length. One can either stop whenever the shorter list has been exhausted, or one can fail (raise and exception) when the lists are of different lengths. Since both treatments can be appropriate in particular circumstances, the ListPair provides two versions of each iterator (including the zip function), one implementing each of the two policies for unequal lists. For instance, there is an app function that accepts unequal lists, and appEq that requires it arguments to have the same length.

`app (`*f*`: 'a * 'b -> unit) (`*s1*`: 'a list * `*s2*`: 'a list) : unit`
apply a function *f* to each pair of corresponding elements of the sequences *s1* and *s2* for effect, returning the unit value when either argument list is exhausted

`app (`*f*`: 'a * 'b -> unit) (`*s1*`: 'a list * `*s1*`: 'a list) : unit`
similar to app, but raising `UnequalLengths` when one of *s1* or *s2* is exhausted before the other

5 Two-dimensional arrays

There are no analogs of `ListPair` for arrays and vectors, but there is support for rectangular two-dimensional arrays in the Array2 structure and it's monomorphic variants (`MONO_ARRAY2`).

The Array2 structure provides two-dimensional generalizations of the array constructors array, fromList, and tabulate. It also provides two-dimensional versions of the iterators app and fold (in basic and index-passing forms) and the modifier functions modify and copy. All the iterators (and tabulate) are parameterized on a *traversal* argument, which is a flag that determines whether the traversal of the array argument will be in row-major or column-major order. There is only one variant of fold (and foldi), which always works left to right (i.e. in order of increasing index).

The two-dimensional analog of a slice is called a region, and the index-passing forms of the iterators are defined to operate on regions rather than arrays. In contrast to the slice types, the region type is a concrete record type, so there are no special functions for creating regions.

6 Conclusion

Lists, vectors and arrays provide a choice of representations for finite sequences, each suited for different patterns of construction and use. Lists are the obvious choice when a sequence is constructed incrementally over time, and when the normal pattern of access is sequential. Vectors are more compact and provide efficient random access when a sequence can be constructed at one point. Arrays are used when update in place is required.

Sometimes it is reasonable to construct a sequence using one representation (e.g. lists), and then switch to another representation (e.g. vectors) when accessing elements of the completed list. The cost of creating the new representation can be amortized over the set of accesses.

Here are the signatures describing the sequence modules and the sections in which they are documented.

`signature LIST`	(Section 10.20)
`signature LIST_PAIR`	(Section 10.21)
`signature VECTOR`	(Section 10.65)
`signature MONO_VECTOR`	(Section 10.26)
`signature VECTOR_SLICE`	(Section 10.66)
`signature MONO_VECTOR_SLICE`	(Section 10.27)
`signature ARRAY`	(Section 10.1)
`signature MONO_ARRAY`	(Section 10.23)
`signature ARRAY_SLICE`	(Section 10.3)
`signature MONO_ARRAY_SLICE`	(Section 10.25)
`signature ARRAY2`	(Section 10.2)
`signature MONO_ARRAY2`	(Section 10.24)