Contents

1 Preamble 2

2 Signature 3

3 Documentation 5
  3.1 Constructing a Sequence 6
  3.2 Deconstructing a Sequence 7
  3.3 Simple Transformations 8
  3.4 Combinators and Higher-Order Functions 9
  3.5 Indexing-Related Functions 11
  3.6 Sorting and Searching 13

4 Views 14
  4.1 List Views 14
  4.2 Tree Views 14
1 Preamble

The type `Seq.t` represents sequences. Sequences are *parallel collections*: ordered collections of things, with parallelism-friendly operations on them. Don’t think of sequences as being implemented by lists or trees (though you could implement them as such); think of them as a new built-in type with only the operations we’re about to describe. The differences between sequences and lists or trees is the cost of the operations, which we specify below. In this document, we describe the cost of array-based sequences.
2 Signature

signature SEQUENCE =

sig

  type 'a t
  type 'a seq = 'a t

exception Range of string

(* Constructing a Sequence *)

val empty : unit -> 'a seq
val singleton : 'a -> 'a seq
val tabulate : (int -> 'a) -> int -> 'a seq
val fromList : 'a list -> 'a seq

(* Deconstructing a Sequence *)

val nth : 'a seq -> int -> 'a
val null : 'a seq -> bool
val length : 'a seq -> int
val toList : 'a seq -> 'a list
val toString : ('a -> string) -> 'a seq -> string
val equal : ('a * 'a -> bool) -> 'a seq * 'a seq -> bool

(* Simple Transformations *)

val rev : 'a seq -> 'a seq
val append : 'a seq * 'a seq -> 'a seq
val flatten : 'a seq seq -> 'a seq
val cons : 'a -> 'a seq -> 'a seq

(* Combinators and Higher-Order Functions *)

val filter : ('a -> bool) -> 'a seq -> 'a seq
val map : ('a -> 'b) -> 'a seq -> 'b seq
val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
val reduce1 : ('a * 'a -> 'a) -> 'a seq -> 'a
val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
val zip : ('a seq * 'b seq) -> ('a * 'b) seq
val zipWith : ('a * 'b -> 'c) -> 'a seq * 'b seq -> 'c seq
(* Indexing-Related Functions *)

val enum : 'a seq -> (int * 'a) seq
val mapIdx : (int * 'a -> 'b) -> 'a seq -> 'b seq
val update : ('a seq * (int * 'a)) -> 'a seq
val inject : 'a seq * (int * 'a) seq -> 'a seq

val subseq : 'a seq -> int * int -> 'a seq
val take : 'a seq -> int -> 'a seq
val drop : 'a seq -> int -> 'a seq
val split : 'a seq -> int -> 'a seq * 'a seq

(* Sorting and Searching *)

val sort : ('a * 'a -> order) -> 'a seq -> 'a seq
val merge : ('a * 'a -> order) -> 'a seq * 'a seq -> 'a seq
val search : ('a * 'a -> order) -> 'a -> 'a seq -> int option

(* Views *)

datatype 'a lview = Nil | Cons of 'a * 'a seq
val showl : 'a seq -> 'a lview
val hidel : 'a lview -> 'a seq

datatype 'a tview = Empty | Leaf of 'a | Node of 'a seq * 'a seq
val showt : 'a seq -> 'a tview
val hidet : 'a tview -> 'a seq

end
3 Documentation

We assume that all functions that are given as arguments (such as the f in map f) have $O(1)$ work and span. In order to analyze the runtime of sequence functions when this is not the case, we need to analyze the corresponding cost graphs.

**Constraint:** Whenever you use these sequence functions, please make sure you meet the specified preconditions.

If you do not meet the precondition for a function, it may not behave as expected or meet the cost bounds stated below.

**Definition 1 (Associative).** Fix some type $t$. We say a function $g : t \times t \rightarrow t$ is associative if for all $a$, $b$, and $c$ of type $t$:

$$g(g(a,b),c) \equiv g(a,g(b,c))$$

**Definition 2 (Identity).** Fix some type $t$. Given a function $g : t \times t \rightarrow t$, we say $z$ is the identity for $g$ if for all $x : t$:

$$g(x,z) = g(z,x) = x$$
3.1 Constructing a Sequence

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Requires</th>
<th>Ensures</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>empty</code></td>
<td><code>unit -&gt; 'a seq</code></td>
<td></td>
<td><code>empty ()</code> evaluates to the empty sequence (the sequence of length zero).</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td><code>singleton</code></td>
<td><code>'a -&gt; 'a seq</code></td>
<td></td>
<td><code>singleton x</code> evaluates to a sequence of length 1 whose only element is <code>x</code></td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td><code>tabulate</code></td>
<td><code>(int -&gt; 'a) -&gt; int -&gt; 'a seq</code></td>
<td><code>For all 0 \leq i &lt; n, f i is valuable.</code></td>
<td><code>tabulate f n</code> evaluates to a sequence (S) of length (n), where the (i^{th}) element of (S) is equal to (f i). Note that indices are zero-indexed. Raises <code>Range</code> if (n) is less than zero.</td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td><code>fromList</code></td>
<td><code>'a list -&gt; 'a seq</code></td>
<td></td>
<td><code>fromList L</code> returns a sequence consisting of the elements of (L), preserving order. This function is intended primarily for debugging purposes.</td>
<td>(O(</td>
<td>L</td>
</tr>
</tbody>
</table>
3.2 Deconstructing a Sequence

nth : 'a seq -> int -> 'a
ENSURES: nth S i evaluates to the \(i^{th}\) element (zero-indexed) of \(S\). Raises Range if \(i\) is negative or greater than \((\text{length } S) - 1\).
Work \(O(1)\), Span \(O(1)\).

null : 'a seq -> bool
ENSURES: null S evaluates to true if \(S\) is an empty sequence, and false otherwise.
Work \(O(1)\), Span \(O(1)\).

length : 'a seq -> int
ENSURES: length S (often written as |\(S|\)) evaluates to the number of items in \(S\).
Work \(O(1)\), Span \(O(1)\).

toList : 'a seq -> 'a list
ENSURES: toList S returns a list consisting of the elements of \(S\), preserving order. This function is intended primarily for debugging purposes.
Work \(O(|S|)\), Span \(O(|S|)\).

toString : ('a -> string) -> 'a seq -> string
REQUIRES: ts x evaluates to a value for all elements \(x\) of \(S\) (e.g. if \(ts\) is total).
ENSURES: toString ts S evaluates to a string representation of \(S\), using the function \(ts\) to convert each element of \(S\) into a string.
Work \(O(|S|)\), Span \(O(\log |S|)\).

equal : ('a * 'a -> bool) -> 'a seq * 'a seq -> bool
REQUIRES: eq is total.
ENSURES: equal eq \((S1,S2)\) returns whether or not the two sequences are equal according to the equality function \(eq\).
Work \(O(\min(|S1|,|S2|))\), Span \(O(\log(\min(|S1|,|S2|)))\).
3.3 Simple Transformations

\textbf{rev} : 'a seq -> 'a seq

ENSURES: \texttt{rev} S returns the sequence containing the elements of S in reverse order.

Work $O(|S|)$, Span $O(1)$.

\textbf{append} : 'a seq * 'a seq -> 'a seq

ENSURES: \texttt{append} (S1, S2) evaluates to a sequence of length $|S1| + |S2|$ whose first $|S1|$ elements are the sequence S1 and whose last $|S2|$ elements are the sequence S2.

Work $O(|S1| + |S2|)$, Span $O(1)$.

\textbf{flatten} : 'a seq seq -> 'a seq

ENSURES: \texttt{flatten} S flattens a sequence of sequences down to a single sequence (similar to List.concat for lists).

Work $O\left(|S| + \sum_{s \in S} |s|\right)$, Span $O(\log |S|)$.

\textbf{cons} : 'a -> 'a seq -> 'a seq

ENSURES: If the length of S is n, \texttt{cons} x S evaluates to a sequence of length n+1 whose first item is x and whose remaining n items are exactly the sequence S.

**Constraint**: Beware of using \texttt{cons}. When overused, it often leads to a sequential coding style with little opportunity for parallelism, defeating the purpose of using sequences. See if there’s a way to use other sequence functions to achieve your objective.

Work $O(|S|)$, Span $O(1)$. 
### 3.4 Combinators and Higher-Order Functions

**filter :** \( ('a \to \text{bool}) \to 'a\ seq \to 'a\ seq \)**  
REQUIRES: \( p\ x \) evaluates to a value for all elements \( x \) of \( S \) (e.g. if \( p \) is total)  
ENSURES: \( \text{filter } p\ S \) evaluates to a sequence containing all of the elements \( x \) of \( S \) such that \( p\ x \Rightarrow \text{true} \), preserving element order.  
Work \( O(|S|) \), Span \( O(\log |S|) \).  

**map :** \( ('a \to 'b) \to 'a\ seq \to 'b\ seq \)**  
REQUIRES: \( f\ x \) evaluates to a value for all elements \( x \) of \( S \) (e.g. if \( f \) is total).  
ENSURES: \( \text{map } f\ S \Rightarrow S' \) such that \( |S| = |S'| \) and for all \( 0 \leq i < |S'| \), \( \text{nth } S'\ i \equiv f\ (\text{nth } S\ i) \).  
Work \( O(|S|) \), Span \( O(1) \).  

**reduce :** \( ('a \times 'a \to 'a) \to 'a\ -> 'a\ seq \to 'a \)**  
REQUIRES:  
- \( g \) is total and associative.  
- \( z \) is the identity for \( g \).  
ENSURES: \( \text{reduce } g\ z\ S \) uses the function \( g \) to combine the elements of \( S \) using \( z \) as a base case (analogous to \( \text{foldr } g\ z\ L \) for lists, but with a less-general type).  
Work \( O(|S|) \), Span \( O(\log |S|) \).  

**reduce1 :** \( ('a \times 'a \to 'a) \to 'a\ seq \to 'a \)**  
REQUIRES: \( g \) is total and associative.  
ENSURES: \( \text{reduce1 } g\ S \) uses the function \( g \) to combine the elements of \( S \). If \( S \) is a singleton sequence, the sequence element is returned. Raises \text{Range} if \( S \) is empty.  
Work \( O(|S|) \), Span \( O(\log |S|) \).  

**mapreduce :** \( ('a \to 'b) \to 'b\ -> ('b \times 'b \to 'b) \to 'a\ seq \to 'b \)**  
REQUIRES: \( g \) and \( f \) meet the preconditions of \( \text{reduce} \) and \( \text{map} \), respectively.  
ENSURES: \( \text{mapreduce } f\ z\ g\ S \equiv \text{reduce } g\ z\ (\text{map } f\ S) \)  
Work \( O(|S|) \), Span \( O(\log |S|) \).
zip : 'a seq * 'b seq -> ('a * 'b) seq

ENSURES: zip (S1,S2) evaluates to a sequence of length min(length S1,length S2) whose i\textsuperscript{th} element is the pair of the i\textsuperscript{th} element of S1 and the i\textsuperscript{th} element of S2.

Work O(min(|S1|,|S2|)), Span O(1).

zipWith : ('a * 'b -> 'c) -> 'a seq * 'b seq -> 'c seq

ENSURES: zipWith f (S1,S2) \equiv map f (zip (S1,S2)).

Work O(min(|S1|,|S2|)), Span O(1).
### 3.5 Indexing-Related Functions

```plaintext
enum : 'a seq -> (int * 'a) seq
ENSURES: enum S evaluates to a sequence such that for each index 0 ≤ i < length S, the i\textsuperscript{th} index of the result is (i, nth S i).
Work O(|S|), Span O(1).

mapIdx : (int * 'a -> 'b) -> 'a seq -> 'b seq
ENSURES: mapIdx f S ≃ map f (enum S).
Work O(|S|), Span O(1).

update : 'a seq * (int * 'a) -> 'a seq
ENSURES: update (S,(i,x)) returns a sequence identical to S but with the i\textsuperscript{th} element (0-indexed) now x if 0 ≤ i < length(S), and raises Range otherwise.
Work O(|S|), Span O(1).

inject : 'a seq * (int * 'a) seq -> 'a seq
ENSURES: inject (S,U) evaluates to a sequence where for each (i,x) in U, the i\textsuperscript{th} element of S is replaced with x. If there are multiple elements at the same index, one is chosen nondeterministically. If any indices are out of bounds, raises Range.
Work O(|S| + |U|), Span O(1).

subseq : 'a seq -> int * int -> 'a seq
ENSURES: subseq S (i,l) takes the subsequence of S with length l starting at index i. If any indices are out of bounds, raises Range.
Work O(1), Span O(1).

take : 'a seq -> int -> 'a seq
ENSURES: take S i evaluates to the sequence containing exactly the first i elements of S if 0 ≤ i ≤ length S, and raises Range otherwise.
Work O(1), Span O(1).

drop : 'a seq -> int -> 'a seq
ENSURES: drop S i evaluates to the sequence containing all but the first i elements of S if 0 ≤ i ≤ length S, and raises Range otherwise.
Work O(1), Span O(1).
```
split : 'a seq -> int -> 'a seq * 'a seq

ENSURES: split S i evaluates to a pair of sequences (S1,S2) such that S1 has length i and append (S1,S2) ≡ s if 0 ≤ i ≤ length S, and raises Range otherwise.

Work O(1), Span O(1).
3.6 Sorting and Searching

```
sort : ('a * 'a -> order) -> 'a seq -> 'a seq
REQUIRES: cmp is total.
ENSURES: sort cmp S returns a permutation of S that is sorted according to cmp.
The sort is stable: elements that are considered equal by cmp remain in the same order they were in S.
Work O(|S| log |S|), Span O(log^2 |S|).
```

```
merge : ('a * 'a -> order) -> 'a seq * 'a seq -> 'a seq
REQUIRES:
  • S1 and S2 are both cmp-sorted.
  • cmp is total.
ENSURES: merge cmp (S1, S2) returns a sorted permutation of append (S1, S2).
  Work O(|S1| + |S2|), Span O(log(|S1| + |S2|)).
```

```
search : ('a * 'a -> order) -> 'a -> 'a seq -> int option
REQUIRES:
  • cmp is total.
  • S is cmp-sorted.
ENSURES: search cmp x S ⇒ SOME i where i is the first index in S satisfying cmp (nth i S, x) ≡ EQUAL or NONE if no such index exists.
  Work O(log |S|), Span O(log |S|).
```
4 Views

4.1 List Views

Recall that list operations have bad parallel complexity, whereas the corresponding sequence operations are much better.

However, sometimes you want to write a sequential algorithm (e.g., because the inputs aren’t very big, or because no good parallel algorithms are known for the problem). Given the sequence interface so far, it is difficult to decompose a sequence as “either empty, or a cons with a head and a tail.” You could write this jank codemonkey’s “length induction instead of structural induction”...

```
case Seq.length s of
  0 =>
  _ => ... uses (Seq.hd s) and (Seq.tl s) ...
```

But nah. We can solve this problem using a view. We’ll put an appropriate datatype in the signature, along with corresponding functions that convert sequences to and from this datatype. This allows us to pattern-match on an abstract type, while keeping the actual representation abstract. These definitions enable viewing a sequence like a list:

```
data datatype 'a lview = Nil | Cons of 'a * 'a seq
val showl : 'a seq -> 'a lview
val hidel : 'a lview -> 'a seq
```

Note the invariant:

```
showl (hidel v) ⇒ v
```

Because the datatype definition is in the signature, the constructors Nil and Cons can be used outside the abstraction boundary. The showl and hidel functions convert between sequences and list views. The following is an example of using this view to perform list-like pattern matching:

```
case Seq.showl s of
  Seq.Nil => ... (* Nil case *)
| Seq.Cons (x,s') => ... uses x and s' ... (* Cons case *)
```

Note that the second argument to Cons is another 'a seq, not an lview. Thus, showl lets you do one level of pattern matching at a time: you can write patterns like Seq.Cons (x,xs) but not Seq.Cons (x,Seq.Nil) (to match a sequence with exactly one element).

We have also provided hidel, which converts a view back to a sequence—Seq.hidel (Seq.Cons (x,xs)) is equivalent to Seq.cons(x,xs) and Seq.hidel Seq.Nil is equivalent to Seq.empty().

4.2 Tree Views

The analogous 'a tview, showt, and hidet are provided in the signature.