Programming Assignment #8 (SML)

CS–671

due 28 Apr 2013 11:69 PM

This assignment illustrates the idea of *lazy evaluation*. Some programming languages (most notably Haskell) rely heavily on lazy evaluation. SML does not, but lazy evaluation can be simulated to build a datatype of sequences, which are conceptually infinite lists. The signature of the structure is listed in Lis. 1.

The idea of lazy evaluation is to delay the evaluation of an expression until its value is absolutely needed. Some functional languages use it extensively. By contrast, most languages—including SML—use eager evaluation. For instance, to evaluate $F(e)$, SML or Java first evaluate e, then call F. In lazy evaluated languages like Haskell, the call to F starts without evaluating e; within F, e (or parts of e) will be evaluated as needed.

Lazy evaluation makes it possible to implement data structures that are conceptually infinite in size. They are, or course, never fully evaluated. Sequences are an example of such a structure. They represent lists that contain an infinite number of elements.

Even though SML uses eager evaluation, sequences can be implemented as a datatype. The trick is to embed in the datatype a function that represents the part of the structure yet to be evaluated.

1. Write a structure `Seq` that implements the signature `SEQ` given in file `sequence-sig.sml`. This structure implements polymorphic streams (infinite sequences). 100 pts

   The implementation used in this assignment relies on the following strategy. The tail of a sequence is implemented as a function towards the actual tail (lazy evaluation). This function is stored in the datatype and run only when the actual value of the tail is needed. In order to be able to filter out all elements from a sequence, the head is implemented as an option: `SOME(x)` means that the actual head is `x`; `NONE` means that the actual head is further down in the sequence.

   This results in a datatype of the form:

   ```
   datatype 'a seq = Cons of 'a option * (unit -> 'a seq)
   ```

   (The name of the constructor (`Cons`) is unimportant; it is not exported in the signature.) The `hd` function can be written this way:

   ```
   fun hd (Cons(NONE, f)) = hd (f()) (* nothing found, keep digging *)
    |  hd (Cons(SOME x, _)) = x      (* found actual head x *)
   ```

   Note how these sequences are conceptually infinite: They always have a tail (no `nil` like with lists). What makes this possible is the fact that the tail is evaluated only when needed, as in the first branch of the `hd` function above.

   The structure to implement contains the following elements. In this description, $[x_1, x_2, \cdots, x_n]$ represents a list and $\langle x_1, x_2, \cdots\rangle$ represents a sequence.

   - `type 'a seq`:
     It can be implemented as the datatype above.
   - `hd: 'a seq -> 'a`:
     It can be implemented as above: `hd`($\langle x_1, x_2, \cdots\rangle$) is x_1.
   - `tl: 'a seq -> 'a seq`:
     Returns a sequence with the first element removed: `tl`($\langle x_1, x_2, \cdots\rangle$) is $\langle x_2, \cdots\rangle$.

```
signature SEQ = sig

type 'a seq

val hd : 'a seq -> 'a
val tl : 'a seq -> 'a seq
val take : 'a seq * int -> 'a list
val drop : 'a seq * int -> 'a seq
val append : 'a list * 'a seq -> 'a seq

val map : ('a -> 'b) -> 'a seq -> 'b seq
val filter : ('a -> bool) -> 'a seq -> 'a seq
val find : int -> ('a -> bool) -> 'a seq -> 'a option

val tabulate : (int -> 'a) -> 'a seq
val iter : ('a -> 'a) -> 'a -> 'a seq
val iterList : ('a list -> 'a) -> 'a list -> 'a seq
val repeat : 'a list -> 'a seq

val merge : 'a seq * 'a seq -> 'a seq
val mergeList1 : 'a seq list -> 'a seq
val mergeList2 : 'a list seq -> 'a seq
val mergeSeq : 'a seq seq -> 'a seq

val allLists : 'a list -> 'a list seq

val Naturals : int seq
val upTo : int -> int seq
val Primes : int seq

val randomInt : int -> int seq
val randomReal: int -> real seq
end
```

Listing 1: `SEQ` signature.

- `take: ’a seq * int -> ’a list`:
  This is a generalization of `List.take`: `take`$(\langle x_1, x_2, \cdots\rangle, k)$ is $[x_1, x_2, \cdots, x_k]$.
- `drop: ’a seq * int -> ’a seq`:
  This is a generalization of `List.drop`: `drop`$(\langle x_1, x_2, \cdots\rangle, k)$ is $\langle x_{k+1}, x_{k+2}, \cdots\rangle$.
- `append: ’a list * ’a seq -> ’a seq`:
  This is a generalization of `List.@`: `append`$([x_1, x_2, \cdots, x_n], \langle y_1, y_2, \cdots\rangle)$ is $\langle x_1, x_2, \cdots, x_n, y_1, y_2, \cdots\rangle$.
- `map: (’a -> ’b) -> ’a seq -> ’b seq`:
  This is a generalization of `List.map`: `map` F $\langle x_1, x_2, \cdots\rangle$ is $\langle F(x_1), F(x_2), \cdots\rangle$.
- `filter: (’a -> bool) -> ’a seq -> ’a seq`:
  This is a generalization of `List.filter`: `filter` F S is the subsequence of S of elements x such that $F(x)$ is true.
- `find: int -> (’a -> bool) -> ’a seq -> ’a option`:
  This is a generalization of `List.find`: `find` N F S is the first element x of S such that $F(x)$ is true, as an option. If no such element is found within the first N values of the sequence, the function returns `NONE`.
- `tabulate: (int -> ’a) -> ’a seq`:
  This is a generalization of `List.tabulate`: `tabulate` F is $\langle F(0), F(1), F(2), F(3), \cdots\rangle$.
- `iter: (’a -> ’a) -> ’a -> ’a seq`:
  This is another way to build a sequence from a function: `iter` F x is $\langle x, F(x), F(F(x)), \cdots\rangle$.

- `iterList: ('a list -> 'a) -> 'a list -> 'a seq`:
  This is a generalization of `iter` in which function F is applied to the previous n elements (when $n = 1$, `iterList` is the same thing as `iter`):

  `iterList` $F\,[x_1, x_2, \cdots, x_n]$ is $\langle x_1, x_2, \cdots, x_n, F([x_1, x_2, \cdots, x_n]), F([x_2, \cdots, x_n, F([x_1, x_2, \cdots, x_n])]),$
  $F([x_3, \cdots, x_n, F([x_1, x_2, \cdots, x_n]), F([x_2, \cdots, x_n, F([x_1, x_2, \cdots, x_n])])]), \cdots\rangle$.

  For instance, `iterList (fn [x,y] => x+y) [0,1]` is the sequence of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... `Iterlist(f)` raises `Empty` if called on an empty list.

- `repeat: 'a list -> 'a seq`:
  This is a way to build a sequence by repeating a list: `repeat` $[x_1, x_2, ..., x_n]$ is $\langle x_1, \cdots, x_n, x_1, \cdots, x_n, x_1, \cdots\rangle$. The function raises `Empty` if its input is empty.

- `merge: 'a seq * 'a seq -> 'a seq`:
  This merges two sequences into a single sequence: `merge`$(\langle x_1, x_2, \cdots\rangle, \langle y_1, y_2, \cdots\rangle)$ is $\langle x_1, y_1, x_2, y_2, \cdots\rangle$.

- `mergeList1: 'a seq list -> 'a seq`:
  This is a generalization of merge: It merges a list of sequences into a single sequence: `mergeList1` $[\langle x_1, x_2, \cdots\rangle, \langle y_1, y_2, \cdots\rangle, \cdots]$ is $\langle x_1, y_1, \cdots, x_2, y_2, \cdots\rangle$. The function raises `Empty` if its input is empty.

- `mergeList2: 'a list seq -> 'a seq`:
  This is a generalization of merge: It merges a sequence of lists into a single sequence: `mergeList2` $\langle [x_1, x_2, \cdots, x_{n_1}], [y_1, y_2, \cdots, y_{n_2}], \cdots\rangle$ is $\langle x_1, x_2, \cdots, x_{n_1}, y_1, y_2, \cdots, y_{n_2}, \cdots\rangle$.

- `mergeSeq: 'a seq seq -> 'a seq`:
  This is a generalization of merge: It merges a sequence of sequences into a single sequence: `mergeSeq` $\langle s_1, s_2, \cdots\rangle$ is a sequence that contains *all* the elements of s_1 exactly once, all the elements of s_2 exactly once, etc.[1] The order in which these elements appear is not specified. Note that the resulting sequence does not consist of all the elements of s_1 followed by all the elements of s_2, etc., since all the sequences are infinite in length.

- `allLists: 'a list -> 'a list seq`:
  `allLists` L is a sequence that contains *all* the lists that can be made from the elements of L. Each possible list appears exactly once. The order in which the lists appear is not specified. Note that lists are ordered structures, potentially with duplicates. Therefore, `allLists [0,1]` contains an infinite number of lists, including `[]`, `[1,1,0,1]`, `[1,1,1]`, `[0,1,1,1]`, `[0,0,0,0,0,0,0,0,0]`, ...

  This function is optional and is left as a bonus question.

- `Naturals: int seq`:
  This is the sequence $\langle 0, 1, 2, \cdots\rangle$.

- `upTo: int -> int seq`:
  `upTo` N is the sequence $\langle 0, 1, 2, \cdots, N-1, N, N, N, \cdots\rangle$.

- `Primes: int seq`:
  This is the sequence of prime numbers: $\langle 2, 3, 5, 7, 11, 13, 17, 19, \cdots\rangle$. This sequence can be constructed from the *sieve of Eratosthenes*:

  (a) Start with the sequence $\langle 2, 3, 4, 5, 6, \cdots\rangle$.
  (b) Keep the first number x: it is prime; Let L be the tail of the sequence.
  (c) Remove all multiples of x from L.
  (d) Recursively apply the algorithm from step 2 to L.

  Function `filter` can be used to remove the multiples of x from L. Of course, the recursive call to *sieve* has to be delayed (i.e., made lazily) to avoid a non-terminating recursion.

- `randomInt: int -> int seq`:
  This is a sequence of pseudo-random numbers. It is built using the algorithm from `java.util.Random`,

[1] "Exactly once" means that each element of s_1 appears once in the final result, but if s_1 contains duplicates, these values will appear as many times in the result as they do in s_1.

```java
public class Random { // simplified version

  private long seed;

  private final static long multiplier = 0x5DEECE66DL;
  private final static long addend = 0xBL;
  private final static long mask = (1L << 48) - 1;

  public Random (long seed) {
    this.seed = (seed ^ multiplier) & mask;
  }

  protected int next (int bits) {
    seed = (seed * multiplier + addend) & mask;
    return (int)(seed >>> (48 - bits));
  }

  public int nextInt() {
    return next(32);
  }

  public double nextDouble() {
    return (((long)(next(26)) << 27) + next(27)) / (double)(1L << 53);
  }
}
```

Listing 2: `java.util.Random` (simplified).

listed in Lis. 2. This class implements a linear congruence. The entire state of the pseudo-random generator is 48 bits and is stored in a `long` value (64 bits). The next state is calculated from the current state by multiplying by `0x5DEECE66D`, adding `0xB` and keeping the lowest 48 bits. When n random bits are needed, the highest n bits of the state are returned. (The generator never uses more than 32 bits from its 48-bit state.)

Note that SMLNJ's `int` is a 31-bit signed integer. So, the sequence will contain values equal to what Java would produce with calls to `next(31)`.

SMLNJ has a structure `Word64` that implements 64-bit words with standard arithmetic, logic and shifting operations. This structure can be used to store the state of the pseudo-random generator.

- `randomReal: int -> real seq`:
  This sequence contains values as if they were returned by calls to `Random.nextDouble` on a Java pseudo-random generator created from the same seed. Note that each floating-point value uses *two* successive states from the generator.

Bonus question: Implement function `allLists`. If the bonus question is not attempted, a function should still be provided so the structure is complete (e.g., `fun allLists _ = raise Fail "Not implemented"`). 10 pts

Notes:

- This assignment <u>must</u> be submitted in a file named `8.sml`. This file can load other files using function `use`, if necessary. The given signature cannot be modified and should not be loaded in `8.sml`.
- One aspect of this exercise is to make sure that functions never evaluate more of a sequence than is actually needed. Even though sequences are potentially infinite, they can become "short" or even empty. For instance, if `N` is the sequence of natural numbers $\langle 0, 1, 2, 3, 4, 5, \cdots \rangle$ and

```
val short = filter (fn x => x<10) N
val empty = filter (fn x => x<0) N
```

  then `short` only has 10 elements and `empty` has none. Therefore, `hd(short)` should return a value but `hd(empty)` will run forever, looking for the first negative natural number, which does not exist. In the same way, `take(short,10)` returns a list but `take(short,11)` runs forever, looking for the 11th natural number that is less than 10.

  In general, things that are not absolutely needed to calculate a value should not be evaluated. For instance, `tl(empty)` should not loop but return a sequence (which is empty). For this reason, the following implementation of `tl` *is incorrect*:

```
fun tl (Cons(NONE, f)) = tl(f())
 |  tl (Cons(SOME _, f)) = f()
```

  It works on non empty sequences but will loop forever on empty ones.

- Some functions are more difficult than others but almost all can be implemented independently and in any order. If a function looks difficult and confusing, skip it, move to the next one and come back to it later. The nature of the assignment is such that the more functions you implement, the easier it gets. For any function that is left unimplemented, you need to provide a dummy function so the structure has the desired signature.
- `randomInt` and `randomReal` can both rely on the same sequence of random states, i.e., of type `Word64.word seq`. Function `randomInt` is then just a call to `map`(F) on this sequence with a well-chosen function F; `randomReal` is a little more complicated because each real value uses two states from the sequence.
- Bit operations in SML are awkward. For instance, the default `Word` structure is `Word32`, not `Word64`. Furthermore, shifting operations are specified in terms of `Word31` values instead of `int`. Finally, some functions, like `Word64.toLarge` are not implemented!

  As a result, converting a 31-bit `Word64` value w to a standard (31-bit) `int` is more complicated than it should be. One way is to shift left then right to get sign extension, then convert to a signed `int`:

```
Word64.toIntX (Word64.~>> ((Word64.<< (w, Word31.fromInt 33)), Word31.fromInt 33))
```

  Another way is to go through strings: `Word31.toIntX (valOf (Word31.fromString (Word64.toString w)))`

  Note also that `>>` in Java is `~>>` in SML and `>>>` in Java is `>>` in SML.

- The first 9 values of `randomInt 2012` are:

  $-811092496, 359748371, -219281478, 717414848, 876580681, 825130012, 979518283, 846006240, -723634745$

  The first 7 values of `randomReal 2012` are:

  $0.622305619233, 0.897889097543, 0.408189689878, 0.456123745351, 0.663031323361, 0.55944578207, 0.124641665352$