Techniques for generating lazy sequences

October 14, 2021

Abstract

In this short note, we will look at three techniques for generating lazy sequences. Each technique is slightly different and could find use in different tasks. This note is not thoroughly proofread, so it is very likely that it contains mistakes and typos. If you find any, do let me know.

Throughout this handout some of the exercises have solutions. Attempt the problem on your own before looking at the solution.

The long and seemingly tedious parts of code/terminal output, are outputs from traces of functions. These can be used to understand if a function is lazy or not truly lazy. If you find these too complicated, you can skip them.

1 Introduction

The main example we will be considering is that of generating all binary strings of length k. Note that a function that returns all such strings is the following:

```
(* Non-lazy way of generating all possible binary strings of length k *)
let rec all_binary_seqs k cur_seq cur all =
    if cur = k then cur_seq :: all
    else all_binary_seqs k (0::cur_seq) (cur+1)
       (all_binary_seqs k (1::cur_seq) (cur+1) all);;
```

Checking for $k = 3$, we get

```
all_binary_seqs 3 [] 0 [];;
- : int list list =
[[0; 0; 0]; [1; 0; 0]; [0; 1; 0]; [1; 1; 0]; [0; 0; 1]; [1; 0; 1]; [0; 1; 1];
 [1; 1; 1]]
```

Throughout this handout we will be using the following convenient functions for sequences,

```
type 'a seq = Nil | Cons of 'a * (unit -> 'a seq);;

let rec to_list = function
  Nil -> []
| Cons(x, xf) -> x :: to_list (xf());;

let rec from_list = function
  [] -> Nil
| x::xs -> Cons(x, fun () -> from_list(xs));;
```

Note: We should be careful, because some implementations although they do return a lazy list, they may not be lazy, in the sense that they do not compute each element of the sequence one by one. For example, the following implementation does not generate all binary strings in a lazy manner,

```
let non_lazy k = from_list (all_binary_seqs k [] 0 []);;
```

This function first computes all binary strings in a list and then converts them to a sequence. This means that even if we only end up using say only two elements of the sequence, all 2^k will be generated.

2 A method to check laziness

If you are unsure if a method if lazy, you can use `#trace` to see where the main bulk of the work is being done. For example in the code above when we trace `all_binary_seqs` and `non_lazy`, we see that the main bulk is in the first call and then each subsequent `tl` takes is almost immediate. In the sections below, we will see three approaches for getting lazy implementations of this function.

```
# #trace all_binary_seqs;;
all_binary_seqs is now traced.
# #trace non_lazy;;
non_lazy is now traced.
# let a1 = non_lazy 2;;
non_lazy <-- 2
all_binary_seqs <-- 2
all_binary_seqs --> <fun>
all_binary_seqs* <-- []
all_binary_seqs* --> <fun>
all_binary_seqs** <-- 0
all_binary_seqs** --> <fun>
all_binary_seqs*** <-- []
all_binary_seqs <-- 2
all_binary_seqs --> <fun>
all_binary_seqs* <-- [1]
all_binary_seqs* --> <fun>
all_binary_seqs** <-- 1
all_binary_seqs** --> <fun>
all_binary_seqs*** <-- []
all_binary_seqs <-- 2
all_binary_seqs --> <fun>
all_binary_seqs* <-- [1; 1]
all_binary_seqs* --> <fun>
all_binary_seqs** <-- 2
all_binary_seqs** --> <fun>
all_binary_seqs*** <-- []
all_binary_seqs*** --> [[1; 1]]
all_binary_seqs <-- 2
all_binary_seqs --> <fun>
all_binary_seqs* <-- [0; 1]
all_binary_seqs* --> <fun>
all_binary_seqs** <-- 2
all_binary_seqs** --> <fun>
all_binary_seqs*** <-- [[1; 1]]
all_binary_seqs*** --> [[0; 1]; [1; 1]]
all_binary_seqs*** --> [[0; 1]; [1; 1]]
all_binary_seqs <-- 2
all_binary_seqs --> <fun>
all_binary_seqs* <-- [0]
all_binary_seqs* --> <fun>
all_binary_seqs** <-- 1
all_binary_seqs** --> <fun>
all_binary_seqs*** <-- [[0; 1]; [1; 1]]
all_binary_seqs <-- 2
all_binary_seqs --> <fun>
all_binary_seqs* <-- [1; 0]
all_binary_seqs* --> <fun>
all_binary_seqs** <-- 2
all_binary_seqs** --> <fun>
all_binary_seqs*** <-- [[0; 1]; [1; 1]]
all_binary_seqs*** --> [[1; 0]; [0; 1]; [1; 1]]
all_binary_seqs <-- 2
all_binary_seqs --> <fun>
all_binary_seqs* <-- [0; 0]
all_binary_seqs* --> <fun>
```

```
all_binary_seqs** <-- 2
all_binary_seqs** --> <fun>
all_binary_seqs*** <-- [[1; 0]; [0; 1]; [1; 1]]
all_binary_seqs*** --> [[0; 0]; [1; 0]; [0; 1]; [1; 1]]
all_binary_seqs*** --> [[0; 0]; [1; 0]; [0; 1]; [1; 1]]
all_binary_seqs*** --> [[0; 0]; [1; 0]; [0; 1]; [1; 1]]
non_lazy --> Cons ([0; 0], <fun>)
val a1 : int list seq = Cons ([0; 0], <fun>)
# let a2 = tl a1;;
val a2 : int list seq = Cons ([1; 0], <fun>)
# let a3 = tl a2;;
val a3 : int list seq = Cons ([0; 1], <fun>)
# let a4 = tl a3;;
val a4 : int list seq = Cons ([1; 1], <fun>)
# let a5 = tl a4;;
val a5 : int list seq = Nil
```

3 Approach 1: Using `getNext()`

In this technique, we define a method `getNext()` which given the current element of the sequence, returns the next one. We should choose `getNext()` so that we can simply and efficiently get the next element of the sequence given the current. We should be careful not to generate the same string twice, otherwise there will be an infinite loop.
For example, for the all binary strings of length k sequence one reasonable ordering is the following

$$[0;0;0] \to [1;0;0] \to [0;1;0] \to [1;1;0] \to [0;0;1] \to \dots,$$

which it is like we are doing $+1$ in the binary representation system.
The following function does exactly this:

```
let rec plus_one = function
| 0::xs -> 1::xs
| 1::xs -> 0::plus_one xs;;
```

This will be our `getNext()` function. By noting that the last element of the sequence will be the all 1s vector, we can generate the sequence of binary strings

```
let rec lazy_all_bin_seqs cur = Cons(cur, fun () ->
   if not (List.mem 0 cur) then Nil (* If we reach the all ones vector we are done.
       *)
   else lazy_all_bin_seqs (plus_one cur));;
```

We can test the correctness of the method:

```
# let ans = to_list (lazy_all_bin_seqs [0;0;0]);;
val ans : int list list =
  [[0; 0; 0]; [1; 0; 0]; [0; 1; 0]; [1; 1; 0]; [0; 0; 1]; [1; 0; 1];
   [0; 1; 1]; [1; 1; 1]]
```

By tracing, we can now see that each call to tail finds a new binary string, so the computation is spread among the items of the sequence:

```
# #trace plus_one;;
plus_one is now traced.
# #trace lazy_all_bin_seqs;;
lazy_all_bin_seqs is now traced.
# let a1 = lazy_all_bin_seqs [0;0;0];;
lazy_all_bin_seqs <-- [0; 0; 0]
lazy_all_bin_seqs --> Cons ([0; 0; 0], <fun>)
val a1 : int list seq = Cons ([0; 0; 0], <fun>)
# let a2 = tl a1;;
plus_one <-- [0; 0; 0]
plus_one --> [1; 0; 0]
lazy_all_bin_seqs <-- [1; 0; 0]
lazy_all_bin_seqs --> Cons ([1; 0; 0], <fun>)
val a2 : int list seq = Cons ([1; 0; 0], <fun>)
```

```
# let a3 = tl a2;;
plus_one <-- [1; 0; 0]
plus_one <-- [0; 0]
plus_one --> [1; 0]
plus_one --> [0; 1; 0]
lazy_all_bin_seqs <-- [0; 1; 0]
lazy_all_bin_seqs --> Cons ([0; 1; 0], <fun>)
val a3 : int list seq = Cons ([0; 1; 0], <fun>)
# let a4 = tl a3;;
plus_one <-- [0; 1; 0]
plus_one --> [1; 1; 0]
lazy_all_bin_seqs <-- [1; 1; 0]
lazy_all_bin_seqs --> Cons ([1; 1; 0], <fun>)
val a4 : int list seq = Cons ([1; 1; 0], <fun>)
# let a5 = tl a4;;
plus_one <-- [1; 1; 0]
plus_one <-- [1; 0]
plus_one <-- [0]
plus_one --> [1]
plus_one --> [0; 1]
plus_one --> [0; 0; 1]
lazy_all_bin_seqs <-- [0; 0; 1]
lazy_all_bin_seqs --> Cons ([0; 0; 1], <fun>)
val a5 : int list seq = Cons ([0; 0; 1], <fun>)
```

Exercise 1 [Unbounded binary strings] Modify the code above to generate a sequence with all binary strings.

We modify the `plus_one` function to append 1 if all elements are 1. Also, in the generation phase, we can avoid the check for whether we have reached the end of the sequence.

```
let rec new_plus_one = function
  [] -> [1]
| 0::xs -> 1::xs
| 1::xs -> 0::new_plus_one xs;;

let rec lazy_all_bin_seqs_unbounded cur = Cons(cur, fun () ->
   lazy_all_bin_seqs_unbounded (new_plus_one cur));;

let a1 = lazy_all_bin_seqs_unbounded [0];;
let a2 = tl a1;;
let a3 = tl a2;;
let a4 = tl a3;;
let a5 = tl a4;;
```

Exercise 2 [Generate all b-ary strings] Write a function that returns a sequence with all strings consisting of elements $0, \ldots, b-1$.

We start by defining the addition $+1$ in base b. Then, the code is very similar to unbounded binary strings.

```
let rec new_plus_one_base b = function
  [] -> [1]
| x::xs ->
  if x = b-1 then 0::new_plus_one_base b xs
  else (x+1)::xs;;

let rec all_b_seqs b cur = Cons(cur, fun () ->
   all_b_seqs b (new_plus_one_base b cur));;

let a1 = all_b_seqs 3 [];;
let a2 = tl a1;;
let a3 = tl a2;;
```

```
let a4 = tl a3;;
let a5 = tl a4;;
```

Exercise 3 [Generate all b-ary strings of length k] Write a function that returns a sequence with all strings of length k consisting of $0, \ldots, b-1$.

```
let rec plus_one_base b = function
  [] -> []
| x::xs ->
  if x = b-1 then 0::plus_one_base b xs
  else (x+1)::xs;;

let rec lazy_all_b_seqs b cur = Cons(cur, fun () ->
   if not (List.exists (fun x -> x < b-1) cur) then Nil
   else lazy_all_b_seqs b (plus_one_base b cur));;

(* All possible orderings of 3 elements from a set of 5 *)
let a1 = lazy_all_b_seqs 5 [0;0;0];;
to_list a1;;
```

Exercise 4 Write a function that returns a sequence with the multiples of 3. Generalise to any $k \in \mathbb{N}$.

4 Approach 2: Using operations on sequences and more primitive sequences

The second approach that we will look at is more useful when generating complicated sequences, but can also be applied to generating all binary strings. We assume that we have implemented the following core functions on sequences:

```
let rec from x = Cons(x, fun () -> from (x+1));;

let rec map_seq f = function
  Nil -> Nil
| Cons(x, xf) -> Cons(f x, fun () -> map_seq f (xf()));;

let rec filter_seq f = function
  Nil -> Nil
| Cons(x, xf) ->
    if f x then Cons(x, fun () -> filter_seq f (xf()))
    else filter_seq f (xf());;
```

Instead of counting in binary we can start with the sequence `from 0` and convert each value to its binary representation. Again this would give $0 \to [0], 1 \to [1], 2 \to [0;1], 3 \to [1;1], 4 \to [0;0;1], \ldots$.

```
let rec to_binary x =
  if x = 0 then [0]
  else if x = 1 then [1]
  else (x mod 2)::to_binary (x / 2);;
```

Now we can generate all binary strings as `map_seq to_binary (from 0)`. Running the following code, we get:

```
# let a1 = map_seq to_binary (from 0);;
val a1 : int list seq = Cons ([0], <fun>)
# let a2 = tl a1;;
val a2 : int list seq = Cons ([1], <fun>)
# let a3 = tl a2;;
val a3 : int list seq = Cons ([0; 1], <fun>)
# let a4 = tl a3;;
val a4 : int list seq = Cons ([1; 1], <fun>)
# let a5 = tl a4;;
val a5 : int list seq = Cons ([0; 0; 1], <fun>)
```

This method is more useful for complicated generation tasks. Here are some examples,

Exercise 5 [Balanced Parentheses] Generate a sequence with all balanced parentheses. (Formally a string s is a balanced parenthesis if it is () or it can be written as (s_1) or $s_1 s_2$ where s_1, s_2 are also balanced parentheses).

Note that balanced parentheses are a subset of the binary strings. So, we can just filter out the binary strings that are not balanced.
An algorithm for checking if a binary string is balanced, is the following: For each position i, keeping the number of unmatched parentheses `cnt_open` up to i:

- If $s_i =$ `'('`, then increment `cnt_open`.
- Otherwise, if $s_i =$ `')'`:
  - If `cnt_open` $= 0$, then we cannot find a matching parenthesis for the current `')'`, so the string is not balanced.
  - Otherwise, decrement `cnt_open`.

In the last step, we should also check that the remaining unmatched parentheses are 0, i.e., `cnt_open` $= 0$.[1]
The code for this algorithm is the following:

```
(* A sequence is balanced if the number of 1s in every prefix of
   the sequence is at least as many as the number of 0s and their
   totals is equal. *)
let rec is_balanced count_open = function
  [] -> count_open = 0 (* In the last index, we want all open parentheses to have
     been matched *)
| 0::xs -> is_balanced (count_open + 1) xs
| 1::xs -> if count_open = 0 then false (* We don't have enough open parentheses to
    match this closed parenthesis. *)
           else is_balanced (count_open - 1) xs;; (* We match the most recent open
               parenthesis to this closed one. *)
```

Hence, we can create a sequence with all balanced strings, using

```
let a1 = filter_seq (fun x -> is_balanced 0 x) (lazy_all_bin_seqs_unbounded [0]);;
let a1 = map_seq (List.map (fun x -> if x = 0 then '(' else ')') ) a1;;
```

Exercise 6 Write a function that returns a sequence with all binary strings that have an occurrence of 000 in them.

Exercise 7 Write a function that returns a sequence with the multiples of 4 and 6. Generalise to any $k_1, k_2 \in \mathbb{N}$.

Exercise 8 Write a function that returns a sequence with all multi-sets of k elements from $0, \ldots, b-1$.

In Exercise 3, we saw how to generate all b-strings of length k. Given these, we want to keep one of say $[1;1;2], [2;1;1], [1;2;1]$. To simplify the implementation we can keep the only the sorted list. Hence, we filter out any list that is not sorted.

```
(* All possible orderings of 3 elements (without replacement) from a set of 5 *)
let rec is_unique_aux prev = function
  [] -> true
| x::xs -> if x = prev then false else is_unique_aux x xs;;

let is_unique x = is_unique_aux (-1) (List.sort Int.compare x);;

let a1 = filter_seq is_unique (lazy_all_b_seqs 5 [0;0;0]);;
to_list a1;;
```

[1] We have not rigorously argued that this algorithm works. You will see how to formally argue about algorithms in Part IA Algorithms.

Exercise 9 [All combinations] Write a function that returns a sequence with all combinations of k elements from $0, \ldots, b-1$.

This is the same as the previous exercise, but we don't want to include lists that contain duplicate elements, e.g., we should not keep $[1;2;1]$.

```
(* All unique sets of 3 elements from a set of 5 *)
let rec is_unique_and_sorted_aux prev = function
  [] -> true
| x::xs -> if x <= prev then false else is_unique_and_sorted_aux x xs;;

let is_unique_and_sorted = is_unique_and_sorted_aux (-1);;

let a1 = filter_seq is_unique_and_sorted (lazy_all_b_seqs 5 [0;0;0]);;
to_list a1;;
```

Exercise 10 [Generate all permutations] Write a function that returns a sequence with all permutations of $0, \ldots, n-1$.

We can generate all permutations with n elements, by starting with n zeros and calling `lazy_all_b_seqs n`, and then filtering with the `is_unique` function (defined above).

```
let permutations = filter_seq is_unique (lazy_all_b_seqs 4 [0;0;0;0]);;
to_list permutations;;
```

Note: This approach is quite wasteful because it generates n^n strings and ends up using $n!$. It is possible to do it more efficiently using Approach 1, but it is more challenging. If you are interested search for "next permutation algorithm".

Exercise 11 [Generating all derangements] A derangement is a permutation p where for each i, $p_i \neq i$, i.e., no element maps to itself. (When you are organising a secret Santa event, this is the kind of thing that you want). Write a function that returns a sequence with all derangements.

```
let rec is_derangement i = function
  [] -> true
| x::xs -> if x = i then false
           else is_derangement (i+1) xs;;

let derangemenets = filter_seq (is_derangement 0) permutations;;
to_list derangemenets;;
```

Exercise 12 [Connection to enumerations] *(Attempt after covering enumeration in Discrete Maths)*

(a) Explain how given an enumeration $f : \mathbb{N} \to S$ can be used to generate a sequence (i) if f is an injection and (ii) if it is not an injection.

(b) By defining an injective enumeration $f : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, explain how you can "concatenate" an infinite number of infinite lists.

5 Approach 3: Continuation passing style

The term *continuation passing style* is a term used in programming language design, where the next part of an execution is stored as a function. Whenever this function is called, the rest of the execution will proceed. *(It is possible that you understand what follows without understanding the last two sentences)*
Now, we will try to convert the brute force non-lazy code into a lazy implementation. We start with the original code:

```
let rec all_binary_seqs k cur_seq cur all =
  if cur = k then cur_seq :: all
```

```
  else all_binary_seqs k ([0]::cur_seq) (cur+1)
     (all_binary_seqs k ([1]::cur_seq) (cur+1) all);;
```

We start by replacing `all` with a sequence. In this way we get,

```
let rec all_binary_seqs_3_A k cur_seq cur all =
  if cur = k then Cons(cur_seq, fun () -> all)
  else all_binary_seqs_3_A k (0::cur_seq) (cur+1)
     (all_binary_seqs_3_A k (1::cur_seq) (cur+1) all);;
```

Are we ready? No, not yet. Although the code returns a lazy sequence, when tracing `all_binary_seqs_3_A`, we notice that almost all of the work is happening in the first call.

```
# let a1 = all_binary_seqs_3_A 2 [] 0 Nil;;
all_binary_seqs_3_A <-- 2
all_binary_seqs_3_A --> <fun>
all_binary_seqs_3_A* <-- []
all_binary_seqs_3_A* --> <fun>
all_binary_seqs_3_A** <-- 0
all_binary_seqs_3_A** --> <fun>
all_binary_seqs_3_A*** <-- Nil
all_binary_seqs_3_A <-- 2
all_binary_seqs_3_A --> <fun>
all_binary_seqs_3_A* <-- [1]
all_binary_seqs_3_A* --> <fun>
all_binary_seqs_3_A** <-- 1
all_binary_seqs_3_A** --> <fun>
all_binary_seqs_3_A*** <-- Nil
all_binary_seqs_3_A <-- 2
all_binary_seqs_3_A --> <fun>
all_binary_seqs_3_A* <-- [1; 1]
all_binary_seqs_3_A* --> <fun>
all_binary_seqs_3_A** <-- 2
all_binary_seqs_3_A** --> <fun>
all_binary_seqs_3_A*** <-- Nil
all_binary_seqs_3_A*** --> Cons ([1; 1], <fun>)
all_binary_seqs_3_A <-- 2
all_binary_seqs_3_A --> <fun>
all_binary_seqs_3_A* <-- [0; 1]
all_binary_seqs_3_A* --> <fun>
all_binary_seqs_3_A** <-- 2
all_binary_seqs_3_A** --> <fun>
all_binary_seqs_3_A*** <-- Cons ([1; 1], <fun>)
all_binary_seqs_3_A*** --> Cons ([0; 1], <fun>)
all_binary_seqs_3_A*** --> Cons ([0; 1], <fun>)
all_binary_seqs_3_A <-- 2
all_binary_seqs_3_A --> <fun>
all_binary_seqs_3_A* <-- [0]
all_binary_seqs_3_A* --> <fun>
all_binary_seqs_3_A** <-- 1
all_binary_seqs_3_A** --> <fun>
all_binary_seqs_3_A*** <-- Cons ([0; 1], <fun>)
all_binary_seqs_3_A <-- 2
all_binary_seqs_3_A --> <fun>
all_binary_seqs_3_A* <-- [1; 0]
all_binary_seqs_3_A* --> <fun>
all_binary_seqs_3_A** <-- 2
all_binary_seqs_3_A** --> <fun>
all_binary_seqs_3_A*** <-- Cons ([0; 1], <fun>)
all_binary_seqs_3_A*** --> Cons ([1; 0], <fun>)
all_binary_seqs_3_A <-- 2
all_binary_seqs_3_A --> <fun>
all_binary_seqs_3_A* <-- [0; 0]
all_binary_seqs_3_A* --> <fun>
all_binary_seqs_3_A** <-- 2
```

```
all_binary_seqs_3_A** --> <fun>
all_binary_seqs_3_A*** <-- Cons ([1; 0], <fun>)
all_binary_seqs_3_A*** --> Cons ([0; 0], <fun>)
all_binary_seqs_3_A*** --> Cons ([0; 0], <fun>)
all_binary_seqs_3_A*** --> Cons ([0; 0], <fun>)
val a1 : int list seq = Cons ([0; 0], <fun>)
# let a2 = tl a1::
val a2 : int list seq = Cons ([1; 0], <fun>)
# let a3 = tl a2::
val a3 : int list seq = Cons ([0; 1], <fun>)
# let a4 = tl a3::
val a4 : int list seq = Cons ([1; 1], <fun>)
# let a5 = tl a4::
val a5 : int list seq = Nil
```

What is the problem? The problem is that in the `else` part of the code both calls to `all_binary_seqs_3_A` are unconditionally executed. Hence, when we write `Cons(cur_seq, fun () -> all)`, the `all` argument is already an evaluated sequence.
So, instead of passing a lazy list, we will pass a function that returns a sequence (passing the *continuation*). When we are in the first branch, we just pass `all_fn` to `Cons` meaning that the sequence will fetch more elements only when it has read the current one. When we are in the second branch, (1) we first compute sequences starting with 0, then (2) sequences starting with 1 and finally (3) the ones given by `all_fn`. So we pass in (2) as an `all_fn` to (1) and (3) as an `all_fn` to (2). So, the code becomes,

```
let rec all_binary_seqs_3_B k cur_seq cur all_fn =
  if cur = k then Cons(cur_seq, all_fn)
  else all_binary_seqs_3_B k (0::cur_seq) (cur+1)
     (fun () -> all_binary_seqs_3_B k (1::cur_seq) (cur+1) all_fn);;
```

We can now check that this is indeed a lazy function,

```
# #trace all_binary_seqs_3_B::
all_binary_seqs_3_B is now traced.
# let a1 = all_binary_seqs_3_B 2 [] 0 (fun () -> Nil)::
all_binary_seqs_3_B <-- 2
all_binary_seqs_3_B --> <fun>
all_binary_seqs_3_B* <-- []
all_binary_seqs_3_B* --> <fun>
all_binary_seqs_3_B** <-- 0
all_binary_seqs_3_B** --> <fun>
all_binary_seqs_3_B*** <-- <fun>
all_binary_seqs_3_B <-- 2
all_binary_seqs_3_B --> <fun>
all_binary_seqs_3_B* <-- [0]
all_binary_seqs_3_B* --> <fun>
all_binary_seqs_3_B** <-- 1
all_binary_seqs_3_B** --> <fun>
all_binary_seqs_3_B*** <-- <fun>
all_binary_seqs_3_B <-- 2
all_binary_seqs_3_B --> <fun>
all_binary_seqs_3_B* <-- [0; 0]
all_binary_seqs_3_B* --> <fun>
all_binary_seqs_3_B** <-- 2
all_binary_seqs_3_B** --> <fun>
all_binary_seqs_3_B*** <-- <fun>
all_binary_seqs_3_B*** --> Cons ([0; 0], <fun>)
all_binary_seqs_3_B*** --> Cons ([0; 0], <fun>)
all_binary_seqs_3_B*** --> Cons ([0; 0], <fun>)
val a1 : int list seq = Cons ([0; 0], <fun>)
# let a2 = tl a1::
all_binary_seqs_3_B <-- 2
all_binary_seqs_3_B --> <fun>
all_binary_seqs_3_B* <-- [1; 0]
all_binary_seqs_3_B* --> <fun>
all_binary_seqs_3_B** <-- 2
all_binary_seqs_3_B** --> <fun>
all_binary_seqs_3_B*** <-- <fun>
all_binary_seqs_3_B*** --> Cons ([1; 0], <fun>)
val a2 : int list seq = Cons ([1; 0], <fun>)
```

```
# let a3 = tl a2;;
all_binary_seqs_3_B <-- 2
all_binary_seqs_3_B --> <fun>
all_binary_seqs_3_B* <-- [1]
all_binary_seqs_3_B* --> <fun>
all_binary_seqs_3_B** <-- 1
all_binary_seqs_3_B** --> <fun>
all_binary_seqs_3_B*** <-- <fun>
all_binary_seqs_3_B <-- 2
all_binary_seqs_3_B --> <fun>
all_binary_seqs_3_B* <-- [0; 1]
all_binary_seqs_3_B* --> <fun>
all_binary_seqs_3_B** <-- 2
all_binary_seqs_3_B** --> <fun>
all_binary_seqs_3_B*** <-- <fun>
all_binary_seqs_3_B*** --> Cons ([0; 1], <fun>)
all_binary_seqs_3_B*** --> Cons ([0; 1], <fun>)
val a3 : int list seq = Cons ([0; 1], <fun>)
# let a4 = tl a3;;
all_binary_seqs_3_B <-- 2
all_binary_seqs_3_B --> <fun>
all_binary_seqs_3_B* <-- [1; 1]
all_binary_seqs_3_B* --> <fun>
all_binary_seqs_3_B** <-- 2
all_binary_seqs_3_B** --> <fun>
all_binary_seqs_3_B*** <-- <fun>
all_binary_seqs_3_B*** --> Cons ([1; 1], <fun>)
val a4 : int list seq = Cons ([1; 1], <fun>)
# let a5 = tl a4;;
val a5 : int list seq = Nil
```

Exercise 13 [Lazy enumeration of change (+++)] Code a function to make change using lazy lists, delivering the sequence of all possible ways of making change. Using sequences allows us to compute solutions one at a time when there exists an astronomical number. Represent lists of coins using ordinary lists. (*Hint:* to benefit from laziness you may need to pass around the sequence of alternative solutions as a function of type `unit -> (int list) seq`.)

[Exercise 9.3 in Lecturer's handout]

Following the methodology above, we get

```
let rec all_change amount coins current next =
  if amount = 0 then Cons(current, next)
  else match coins with
      [] -> next()
    | (c::cs) ->
      if c > amount then all_change amount cs current next
      else all_change (amount - c) coins (c::current) (fun () -> all_change amount
          cs current next);;

to_list(all_change 10 [1;2;5] [] (fun () -> Nil));;
```

Exercise 14 [General] Attempt generating any of the sequences from the previous sections using Approach 3.

Exercise 15 [Extension (++)] In both examples, we saw above there were at most two choices in each step (e.g. in the all binary sequences we had to choose between 0 and 1). Is there a generic way to handle taking $b > 2$ choices? As an example problem, you could try generating all sequences of length k with elements in $0, \ldots, b-1$.

6 More

Project 1 [Sequences in Java] Show that the lazy sequence paradigm can be implemented in Java.

Exercise 16 [How many valid answers?] How many valid ways are there to generate a sequence of binary strings of length k? What about generating the sequence of all binary strings?

Exercise 17 Generate a lazy sequence for the prime numbers.