Foundations of Computer Science

Lazy Lists: consuming and joining sequences

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

```
In[1]: type 'a seq =
         | Nil
         | Cons of 'a * (unit -> 'a seq)

Out[1]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Get the first n elements as a list:

```
In[2]: let rec get n s =
         match n, s with
         | 0, _              -> []
         | n, Nil            -> []
         | n, Cons (x, xf) -> x :: get (n-1) (xf ())

Out[2]: val get : int -> 'a seq -> 'a list = <fun>
```

xf () forces evaluation

```
In[1]: type 'a seq =
         | Nil
         | Cons of 'a * (unit -> 'a seq)

Out[1]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Get the first n elements as a list:

```
In[2]: let rec get n s =
         match n, s with
         | 0, _              -> []
         | n, Nil            -> []
         | n, Cons (x, xf) -> x :: get (n-1) (xf ())

Out[2]: val get : int -> 'a seq -> 'a list = <fun>
```

xf () forces evaluation

```
In[1]: type 'a seq =
         | Nil
         | Cons of 'a * (unit -> 'a seq)

Out[1]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Get the first n elements as a list:

```
In[2]: let rec get n s =
         match n, s with
         | 0, _              -> []
         | n, Nil            -> []
         | n, Cons (x, xf) -> x :: get (n-1) (xf ())

Out[2]: val get : int -> 'a seq -> 'a list = <fun>
```

xf () forces evaluation

```
In[1]: type 'a seq =
         | Nil
         | Cons of 'a * (unit -> 'a seq)

Out[1]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Get the first n elements as a list:

```
In[2]: let rec get n s =
         match n, s with
         | 0, _              -> []
         | n, Nil            -> []
         | n, Cons (x, xf) -> x :: get (n-1) (xf ())

Out[2]: val get : int -> 'a seq -> 'a list = <fun>
```

xf () forces evaluation

```
In[1]: type 'a seq =
         | Nil
         | Cons of 'a * (unit -> 'a seq)

Out[1]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Get the first n elements as a list:

```
In[2]: let rec get n s =
         match n, s with
         | 0, _              -> []
         | n, Nil            -> []
         | n, Cons (x, xf) -> x :: get (n-1) (xf ())

Out[2]: val get : int -> 'a seq -> 'a list = <fun>
```

xf () forces evaluation

```
In[1]: type 'a seq =
         | Nil
         | Cons of 'a * (unit -> 'a seq)

Out[1]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Get the first n elements as a list:

```
In[2]: let rec get n s =
         match n, s with
         | 0, _              -> []
         | n, Nil            -> []
         | n, Cons (x, xf) -> x :: get (n-1) (xf ())

Out[2]: val get : int -> 'a seq -> 'a list = <fun>
```

xf () forces evaluation

Sample Evaluation

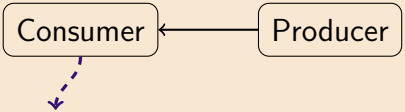
```
let rec from k = Cons (k, fun () -> from (k + 1))

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```

Consumer ← Producer

```
get 2 (from 6)
```

Sample Evaluation

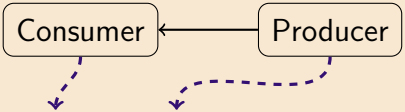
```
let rec from k = Cons (k, fun () -> from (k + 1))

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```




```
get 2 (from 6)
```

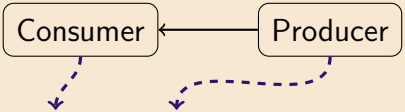
Sample Evaluation

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

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```



Sample Evaluation

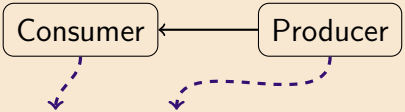
```
let rec from k = Cons (k, fun () -> from (k + 1))

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```




```
get 2 (from 6)  ⇒  get 2 (Cons (6, fun () -> from (6 + 1)))
```

Sample Evaluation

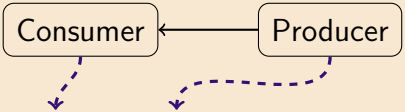
```
let rec from k = Cons (k, fun () -> from (k + 1))

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```




```
get 2 (from 6)  ⇒  get 2 (Cons (6, fun () -> from (6 + 1)))
                ⇒  6 :: get 1 (from (6 + 1))
```

Sample Evaluation

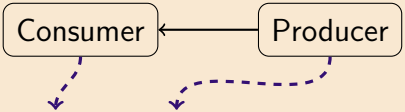
```
let rec from k = Cons (k, fun () -> from (k + 1))

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```




```
get 2 (from 6)  ⇒  get 2 (Cons (6, fun () -> from (6 + 1)))
                ⇒  6 :: get 1 (from (6 + 1))
                ⇒  6 :: get 1 (Cons (7, fun () -> from (7 + 1)))
```

Sample Evaluation

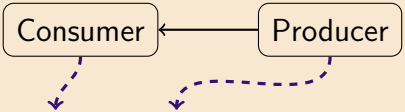
```
let rec from k = Cons (k, fun () -> from (k + 1))

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```




```
get 2 (from 6)  ⇒  get 2 (Cons (6, fun () -> from (6 + 1)))
                ⇒  6 :: get 1 (from (6 + 1))
                ⇒  6 :: get 1 (Cons (7, fun () -> from (7 + 1)))
                ⇒  6 :: 7 :: get 0 (from (7 + 1))
```

Sample Evaluation

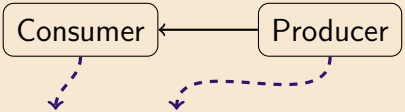
```
let rec from k = Cons (k, fun () -> from (k + 1))

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```




```
get 2 (from 6)  ⇒  get 2 (Cons (6, fun () -> from (6 + 1)))
                ⇒  6 :: get 1 (from (6 + 1))
                ⇒  6 :: get 1 (Cons (7, fun () -> from (7 + 1)))
                ⇒  6 :: 7 :: get 0 (from (7 + 1))
                ⇒  6 :: 7 :: get 0 (Cons (8, fun () -> from (8 + 1)))
```

Sample Evaluation

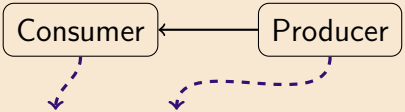
```
let rec from k = Cons (k, fun () -> from (k + 1))

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```




```
get 2 (from 6)  ⇒  get 2 (Cons (6, fun () -> from (6 + 1)))
                ⇒  6 :: get 1 (from (6 + 1))
                ⇒  6 :: get 1 (Cons (7, fun () -> from (7 + 1)))
                ⇒  6 :: 7 :: get 0 (from (7 + 1))
                ⇒  6 :: 7 :: get 0 (Cons (8, fun () -> from (8 + 1)))
                ⇒  6 :: 7 :: []
```

Sample Evaluation

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

let rec get n s =
  match n, s with
  | 0, _              -> []
  | n, Nil            -> []
  | n, Cons (x, xf) -> x :: get (n-1) (xf ())
```




```
get 2 (from 6)  ⇒  get 2 (Cons (6, fun () -> from (6 + 1)))
                ⇒  6 :: get 1 (from (6 + 1))
                ⇒  6 :: get 1 (Cons (7, fun () -> from (7 + 1)))
                ⇒  6 :: 7 :: get 0 (from (7 + 1))
                ⇒  6 :: 7 :: get 0 (Cons (8, fun () -> from (8 + 1)))
                ⇒  6 :: 7 :: []
                ⇒  [6; 7]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```

Joining Two Sequences

```
In[3]:  let rec appendq xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> appendq (xf ()) yq)

Out[3]: val appendq : 'a seq -> 'a seq -> 'a seq = <fun>

In[4]:  get 5 (appendq (from 0) (from 100))

Out[4]: - : int list = [0; 1; 2; 3; 4]
```

A *fair* alternative:

```
In[5]:  let rec interleave xq yq =
          match xq with
          | Nil -> yq
          | Cons (x, xf) -> Cons (x, fun () -> interleave yq (xf ()))

Out[5]: val interleave : 'a seq -> 'a seq -> 'a seq = <fun>

In[6]:  get 5 (interleave (from 0) (from 100))

Out[6]: - : int list = [0; 100; 1; 101; 2]
```