Lec 18: Streams

CS220: Programming Principles

Sang Kil Cha
Streams
Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don’t accept late responses.
Recap: Streams

Stream type: delayed list.

```haskell
type Stream<'a> =
| Nil
| Cons of 'a * (unit -> Stream<'a>)
```

Streams can be used to represent values that are *produced over time*. 
Eliminating “Iterations”

Recall Newton’s method, which is a recursive algorithm for computing a square root.

\[ x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right) \]

```fsharp
define improve guess x = (guess + (x / guess)) / 2.0
define sqrtStream x =
    let rec stream =
        Cons (1.0, fun () -> Stream.map (fun g -> improve g x) stream)
    stream
```
Eliminating States with Streams

Pseudo-Random Number Generator (with a mutable variable).

```ml
let rand seed =
    let mutable r = seed
    fun () ->
        r <- (1103515245 * r + 12345) &&& System.Int32.MaxValue
        r
```
Eliminating States with Streams (cont’d)

Pseudo-Random Number Generator (with stream).

```plaintext
def randStream seed:
    let rec r seed =
        let next = (1103515245 * seed + 12345) &&& System.Int32.MaxValue
        Cons (next, (fun () -> r next))
    r seed
```
BankAccount with Stream?

In essence, we represent time explicitly, using streams, so that we decouple time in our simulated world from the sequence of events that take place during evaluation\(^1\)

\(^1\)Wizard Book Chap. 3.5.5.
let rec bankAccountStream balance amountStream = Cons (balance, 
  fun () ->
    bankAccountStream 
    (balance - Stream.car amountStream)
    (Stream.cdr amountStream))

No mutable state! Therefore, no race condition! We are back to functional.
Memoization
The Performance Problem of Lazy Expression

If we use a delayed object multiple times in a program, it is largely redundant to evaluate the same expression everytime it is referenced.

Key insight to solve the problem: remember the evaluated value and just use it.
Memoization

```ml
let lazyExp () =  
    // complex expressions

let memoizedExp =  
    let mutable v = None  
    fun () ->  
      match v with  
      | None ->  
        let e = lazyExp ()  
        v <- Some e  
        e  
      | Some v -> v
```
Built-in Lazy Expression

let x = lazy 42
x.Force ()

let exp = lazy (printfn "hi"; 42)
exp.Force ()
exp.Force ()

The lazy expression uses memoization internally.
Built-in Stream: Sequence in F#

`seq<'T>` is a stream, we can create a stream with `Seq.unfold` function.

```fsharp
val Seq.unfold: ('State -> ('T * 'State) option) -> 'State -> seq<'T>
```
Infinite Sequence Example

```ml
let ones = Seq.unfold (fun () -> Some (1, ())) ()

let fibs =
    Seq.unfold (fun (a, b) ->
        Some (a, (b, a + b))) (0, 1)

let zeroToInf = Seq.initInfinite (fun n -> n)
```
Finite Sequence Example

```fsharp
let numbers =
    0
    |> Seq.unfold (fun state ->
                    if state > 20 then None
                    else Some(state, state + 1))
```
Unfold Exercise

Write a finite sequence of fibonacci numbers in int32 type, up to the point where the number exceeds the maximum value of int32.
Seq.initInfinite

Write an infinite sequence of fibonacci numbers with Seq.initInfinite.

```ocaml
let rec fibs =  
    Seq.initInfinite (fun n ->  
        if n = 0 then 0  
        elif n = 1 then 1  
        else Seq.item (n - 1) fibs + Seq.item (n - 2) fibs)
```

This is not efficient! Why?
Seq.initInfinite

Write an infinite sequence of fibonacci numbers with Seq.initInfinite.

```ml
let rec fibs =
    Seq.initInfinite (fun n ->
        if n = 0 then 0
        elif n = 1 then 1
        else Seq.item (n - 1) fibs + Seq.item (n - 2) fibs)
```

This is *not* efficient! Why?
Laziness of Sequence

```ocaml
let mySeq = Seq.initInfinite id
let truncatedSeq = Seq.truncate 10 mySeq
let takenSeq1 = Seq.take 10 mySeq
let takenSeq2 = Seq.take 20 truncatedSeq
let printSeq sq = Seq.iter (printf "%d") sq; printfn ""

truncatedSeq |> printSeq
takenSeq1 |> printSeq
takenSeq2 |> printSeq // raise exception here
```
LazyList

What’s the difference between Seq and LazyList?

1. LazyList performs memoization, while Seq does not.
2. LazyList can be pattern-matched directly (with active patterns).
3. LazyList is not a built-in type in F#. It is defined in the FSharpx.Collections library.
Example Usage of LazyList

```fsharp
open FSharpx.Collections

let ones = LazyList.unfold (fun () -> Some (1, ())) ()

match ones with
| LazyList.Cons (n, _) -> printfn "The first element is %d" n
| _ -> printfn "The list is empty"
```
In-Class Activity #18
Preparation

We are going to use the same git repository as before. Just in case you don’t have it, clone the repository using the following command.

1. Clone the repository to your machine.

2. Move in to the directory CS220-Main/Activities
   - cd CS220-Main
   - cd Activities
Problem

Convert the given an infinite LazyList into another LazyList that contains a pairwise sequence of the original list. For example, when the given list is \([1; 2; 3; 4]\), then the output should be \([(1, 2); (3, 4)]\).
Locking
Concurrency Requirement

A concurrent system should produce the same result as if the processes had run sequentially in a certain order. One way to achieve this is to leverage locking primitives, such as mutex.
Mutex (Mutual Exclusion)

Mutex is an object that supports two operations: (1) the mutex can be acquired, and (2) the mutex can be released. Once a mutex is acquired by someone, no other acquire operations on the same mutex can proceed until the mutex is released by the owner.
type Mutex () =
    let mutable lock = false
    member __.TestAndSet () = // This needs H/W support
        if lock then true
        else lock <- true; false

    member __.Acquire () =
        if __.TestAndSet () then __.Acquire () else ()

    member __.Release () =
        lock <- false
Making WithDraw Safe

```ocaml
let makeSerializer =  
  let m = Mutex ()  
  fun p (arg: int) ->  
    m.Acquire ()  
    p arg  
    m.Release ()

let acc = BankAccount (10000)  
let safeWithdraw = acc.Withdraw |> makeSerializer  
safeWithdraw 500  // A  
safeWithdraw 1500  // B  
// Safe even if A and B run concurrently
```
Advanced example.

```csharp
type BankAccount (initial) =
    let m = Mutex ()
    member val Balance = initial with get, set
    member __.WithDraw amount =
        m.Acquire ()
        let newBalance = __.Balance - amount
        __.Balance <- newBalance
        m.Release ()
    member __.Deposit amount =
        m.Acquire ()
        let newBalance = __.Balance + amount
        __.Balance <- newBalance
        m.Release ()
    member __.Transfer amount (account: BankAccount) =
        m.Acquire ()
        __.Balance <- __.Balance - amount
        account.Distribute amount
        m.Release ()
```
Deadlock

Suppose both $A$ and $B$ try to transfer money to each other at the same time.

```javascript
let accA = BankAccount (1000) // A
let accB = BankAccount (500) // B

// Suppose the followings run concurrently
accA.Transfer 100 accB
accB.Transfer 200 accA
```
Locking is Error-Prone

1. When our program has two few locks: data race happens.
2. When our program has too many locks: likely to have deadlocks.

Writing a correct program is extremely difficult with locking!
Stream of Withdrawal

We model the withdrawal processes as a stream of events.

```ml
let balance = 1500

let amountStream = seq [ 1500; 500 ]

let withdrawStream = Seq.unfold (fun (balance, events) ->
  if Seq.isEmpty events then None
  else
    let amount = Seq.head events
    if amount <= balance then
      let newBalance = balance - Seq.head events
      Some (newBalance, (newBalance, Seq.tail events))
    else
      Some (balance, (balance, Seq.tail events))(balance, amountStream)
```
Conclusion
Streams in Practice

1. File I/O.
2. Network sockets.
4. and many more.
Further Readings

- Wizard Book Chap. 3.4 and 3.5.
Question?