Lec 13: Lazy Computation

CS220: Programming Principles

Sang Kil Cha
In-Class Activity #15

In this activity, your goal is to write a member function Perimeter of Triangle and Rectangle. You should not modify the myfunc function. Change only two member functions indicated with FIXME.
Concurrency
Considering the Physical World

Objects in the world do not change one at a time. All the objects in the world act *concurrently*. To model the physical world, it is natural to consider computational processes that execute concurrently.
Price of Mutability = Additional Dimension

An expression of the same symbolic name can have different values at different points in \textit{time}.

\begin{center}
\begin{tcolorbox}
OOP and imperative programming force us to confront \textit{time} as an essential concept in programming.
\end{tcolorbox}
\end{center}
Bank Account Example

Suppose $A$ and $B$ share the same bank account containing 10,000 won. Assume $A$ withdraws 1,500 won, and $B$ withdraws 500 won from the account. What’s the expected balance after the two operations?
Bank Account Example

Suppose $A$ and $B$ share the same bank account containing 10,000 won. Assume $A$ withdraws 1,500 won, and $B$ withdraws 500 won from the account. What’s the expected balance after the two operations?

What if $A$ and $B$ access the same bank account through a network?
type BankAccount (initial) =
    member val Balance = initial with get, set
    member __.WithDraw amount =
        if __.Balance > amount then // 1
            let newBalance = __.Balance - amount // 2
            __.Balance <- newBalance // 3
            printfn "%d won out" amount
        else ()
What is Shared?

Suppose there were two function calls to (`___.Withdraw`) at the same time. Each function call will create its own calling context, and local (in-function) variables will be stored in its calling context, but the property `Balance` will be shared across the two function calls.
Timing Really Matters

A wants to withdraw 1,500 won.

1. if __.Balance > 1500
2. __.Balance - 1500
3. __.Balance = ?

B wants to withdraw 500 won.

4. if __.Balance > 500
5. __.Balance - 500
6. __.Balance = ?

• 1 → 4 → 2 → 5 → 3 → 6.
• 4 → 1 → 2 → 3 → 5 → 6.
• ...
Streams
Back to the Functional World

We’ve learned that OOP and imperative language features are a good tool for modeling real world, but it is at the same time a poor way of handling concurrency. Let’s now go back to our functional world by introducing a new data structure, called streams.
Values Changing Over Time

Why did we need to model a value as an object? Because it changes over time. But, can we model a varying value in a pure functional world?
Values Changing Over Time

Why did we need to model a value as an object? Because it changes over time. But, can we model a varying value in a pure functional world?

Yes. Think of a function $f$ of time $t$: $f(t)$. 
Motivating Example

```
let f =
  let mutable x = 0
  fun () ->
    x <- x + 1
  x
```

```
let f t =
  // "Infinite" list
  let lst = [1; 2; 3; ...]
  lst.[t]
```
Streams

Streams look similar to lists, but it evaluates in a lazy manner.

N.B. F# language provides built-in lazy expressions and features, but we will implement our own first, as we did with List.
Lazy Evaluation

Given an expression, we always eagerly evaluate it in F#. We say F# uses eager evaluation.

```fsharp
let f _ = 
    printfn "body of f"
    true
let g () = 
    printfn "body of g"
    42

g () |> f // What do we see here as a side-effect?
// What if F# was a lazy language?
```
Delaying Evaluation

Although F# is an eager language, we can pretend to be lazy by delaying evaluation of an expression using thunks. A thunk is a function that takes in a unit as input, and returns a value (with or without some side-effects).

```fsharp
let add a b = a + b
let normal = add 1 2 // 3
let delayed = fun () -> add 1 2 // delayed with thunk
delayed () // forcing the delayed expression
```
Stream Implementation

List type.

```haskell
type List<'a> =
  | Nil
  | Cons of 'a * List<'a>
```

Stream type: delayed list.

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

val car: Stream<'a> -> 'a
val cdr: Stream<'a> -> Stream<'a>
val take: Stream<'a> -> int -> Stream<'a> // taking n-first seq
val fromList: 'a list -> Stream<'a>
Infinite Stream

Can we create an infinite stream of ones? \([1; 1; 1; \ldots]\)

```ocaml
let rec ones =  
    Cons (1, fun () -> ones)  

take ones 10 // ?
```

Exercise: can you create a (infinite) stream of positive integers?
Implementing Higher-Order Functions

Implement `map`, `fold`, `filter`, etc. on Stream.
Recursive Values

```ocaml
let rec myval = myval + 1 // error

type BankAccount =
  { mutable Balance: int
    GetBalance: unit -> int }

let rec acc =
  { Balance = 0
    GetBalance = fun () -> acc.Balance } // Delayed
```
Exploiting Streams
In-Class Activity #16

Write a stream of Fibonacci using our own Stream implementation.
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).

```ml
let randStream seed =
    let rec r seed =
        let next = (1103515245 * seed + 12345) &&& System.Int32.MaxValue
        Cons (next, (fun () -> r next))
    r seed
```

```c
// Alternative implementation using System.Random
let random = new System.Random()
let randomStream seed =
    let rec r seed =
        let next = random.Next(seed)
        Cons (next, (fun () -> r next))
    r seed
```
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.
BankAccount with Stream Example

```ocaml
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.
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.
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>
```

```fsharp
let ones = Seq.unfold (fun () -> Some (1, ())) ()
```
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.
Mutex (Conceptual) Implementation

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
```
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.Deposit amount
        m.Release ()
Deadlock

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

```plaintext
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!
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?