Lec 17: Lazy Computation

CS220: Programming Principles

Sang Kil Cha

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 ***time***.

Price of Mutability = Additional Dimension

An expression of the same symbolic name can have different values at different points in ***time***.

> OOP and imperative programming force us to confront ***time*** as an essential concept in programming.

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?

Bank Account Implementation

```
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 = ?`

Assume that the initial balance is 1,500 won. What's the balance after?

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 6$
- $4 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$
- ...

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.

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.

```
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).

```
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
```

Expressing Infinity

We can delay the evaluation of an expression using a function. Can we use this to express an infinite sequence?

Stream Implementation

List type.

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

Stream type: delayed list.

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

In-Class Activity #17

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.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

Problem: Implement Basic Functions for `Stream`

You can raise a `failwith` exception for error cases.

```
val car:  Stream<'a> -> 'a

val cdr:  Stream<'a> -> Stream<'a>

val take:  int -> Stream<'a> -> 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; ...]`

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

take 10 ones // ?
```

Problem: Implement Higher-Order Functions

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

Recursive Values

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

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

let rec acc =
  { Balance = 0
    GetBalance = fun () -> acc.Balance } // Delayed
```

Conclusion

1. Lazy evaluation is a way to delay the evaluation of an expression.
2. Streams are a way to model values that change over time.
3. Streams can be used to model infinite sequences.

Question?