CS3100 - Paradigms of Programming
Final Project

Karan Bardhan(CS20B036) and Smit Bagul(CS20B011)

Abstract

The feature we propose to add in Java is the concept of *force* and *delay*. The basic idea is that the *delay* class returns a so-called promise, which can be redeemed by the *force* class. Thus, the composition of *delay* and *force* carries out a normal evaluation step.
A very important use case of delay and force would be streams. A stream is an infinite list with an element and a promise to evaluate the rest of the stream. This practically allows us to store an infinite amount of data in a finite amount of space. These infinite lists can be used to calculate Armstrong numbers. Another use is that we can implement lazy evaluation in Java by using force and delay in function parameters.

Motivation

Let's say we have an expensive computation in a function parameter that we don't need to compute. Since Java has eager evaluation the computation is made anyway. Using delay would greatly reduce the time and space required for said computation.
For the time being, we define Delay as a class that takes a lambda (or function pointer in languages that support it) and it's argument (let's say Delay takes functions of only arity-1 for now, this can be extended to multiple arity functions by taking a list of arguments as the second argument in Delay) and force as a function that takes a Delay Object and evaluates the function with the given arguments.

```
public class Example
{
    public static void main(String[] args) {
        System.out.println("Hello World");
        Example.OneorInf(true, Example.add(1000000000));
    }
    public static void OneorInf(boolean check, int a) {
        if(check) System.out.println(1);
        else System.out.println(a);
    }
    public static int add(int a) {
        if(a == 0) return 1;
        return add(a-1) + 1;
    }
}
```

This can however be avoided with a delay and force construct. The given code can then be modified as:

```
public class Example
{
    public static void main(String[] args) {
        System.out.println("Hello World");
        Delay<int> D(add, 1000000000)
        Example.OneorInf(true, );
    }
    public static void OneorInf(boolean check, Delay<T> D) {
        if(check) System.out.println(1);
        else System.out.println(force(D));
    }
    public static int add(int a) {
        if(a == 0) return 1;
        return add(a-1) + 1;
    }
}
```

A few drawbacks to this would be:

- We can easily run into a large collection of un-evaluated expressions that could have easily been stored as values, resulting in an unpredictable space behavior.
- It may become harder to analyze code performance.

Semantics

Type-Checking:

- $\frac{<T1>D(<T2>fun(t1,t2,..tm),\{t1',t2'..tn'\})}{T1=T2}$
  Let's say the function fun returns a type T2 and The delay is of type T1, this implies that T1 and T2 are the same, in other words, the delay will be of the type that the function returns

- $\frac{<T1>Delay(<T2>fun(t1,t2,..tn),\{t1',t2'..tn'\})}{ti=ti',\forall i=1..n}$
  Let's say the function fun takes n arguments of type, t1, t2, .., tn, the list passed in the has n elements of type, t1', t2', .., tn', then the types ti must be equal to ti' for all i from 1 to n

- $\frac{<T1>force(<T2>Delay)}{T1=T2}$
  The return type of Force and Delay must be the same

Grammar:
$e ::= e1e2|d|c|force(e)$
$d \in Delay$
$c \in Constant$
Operational Semantics:
$v, w \in Value$
$v ::= d|c$

- $force(d) -> v$
  Force of a delay will result in a value. This value may be a constant or another delay.
- $\frac{e->d \& force(d)->w}{force(e)->w}$
  If an expression reduces to a delay then forcing the exression will give the same result as forcing the reduced delay.

All other semantics will be similar to Java's semantics

Prior Work

Delay and Force have been previously implemented in the Scheme Language. The use cases of delay and force generally exists in most functional programming languages such as Haskell and Scala. For example, although delay and force aren't explicitly defined in Haskell, we can still do lazy evaluation and make infinite lists.

Implementation Plan

Here is a pseudo-code as to how we can implement the force and delay functions in java. func is the lambda that we want to evaluate and args is the list of arguments that the lambda takes.

```
class Delay{
    private:
    func;
    args;
public:
    Delay(func, args){
        this.func = func;
        this.args = args;
    }
    forc(){
        func(args);
    }
}
```

```
class force{
    force(Delay d){
        d.forc();
    }
}
```

Challenges:

- If the arguments are of different types, taking input as a list would be difficult
- We need to change the existing functions to implement delay and force, as in the above example we had to add a force in OneorInf function to implement delay and force concept
- Implementing a delay inside a delay would require extra nesting, which will require more careful implementation by the coder.