Lecture 2: Functional Parallelism

08 August 2020 00:42

Futures: Tasks with Return Values

Functional Programming is when you write your program as pure function call, like

A = F(B)
C = G(A)
D = H(A)

value of A
of B depen
on the valu

① In doesn't matter how man
we call F(B). The value
will always be same.

The value of C & D only
on A. Hence, they can ru
- the ordering is subject

as fn
ds only
ue of B.

---

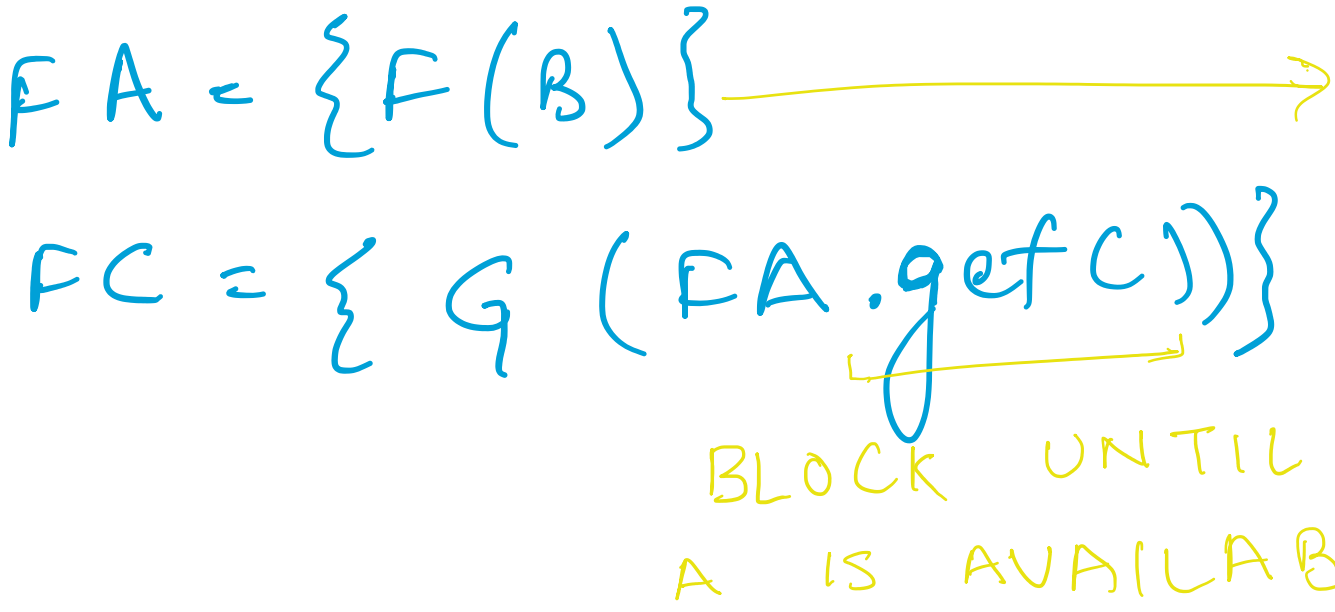
y times
of A

depend
n parallely
to the

constraint that A is avai

So how do we express this parallelism? Well, what we want to do is make an async task. But w
call it a future.
FA = { F (B) }
If we want to use this value of A, for example in calculating C,
C = { F (FA.get()) }

So what is this future object and the get operation? Well, the future refers to what's called a h
object. And in the handle is going to be the final value A. So the difference between FA and A
wrapper for A. And this is something that's returned right away. So when you do future B, it's
task. The program can immediately move to the next statement and the next statement
after that. There is no blocking when you create a future. Now, when you actually need the va
you do the FA dot get, then that's when the task will do a blocking or a wait operation until A



lable.

ve are going to

handle for an
is that FA is a
like an async

alue of A, when
is available.

E

FSUM1= FUTURE{LEFT +

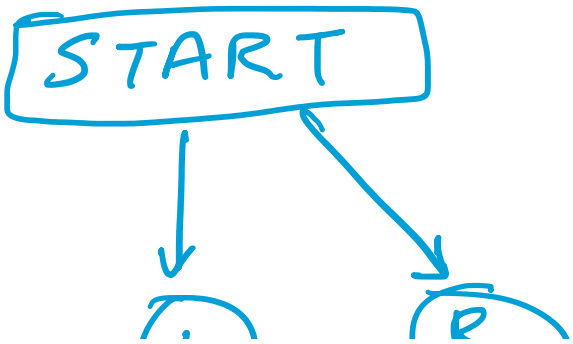
FSUM2= FUTURE{RIGHT

SUM= FSUM1.get()+FSU

Futures in Java's Fork/Join Framework

To implement Futures in Java we use ForkJoin framework. So as before we create a

```
Class SumNumber extends RecursiveTask{
     Int[] A,
     Int low, high
     Compute() {
          If(low == high)
               return A[low];
          Else if( low > high)
               return 0;
          Else {
               Mid = ( low + high ) / 2;
               L= new SumNumber(A, low, mid);
               R = new SumNumber(A, mid+1, high);
               R.fork() // return a future
               Return L.compute() + R.join(); // join implemen
               future.get() and returns the value.
          }
     }
}
```

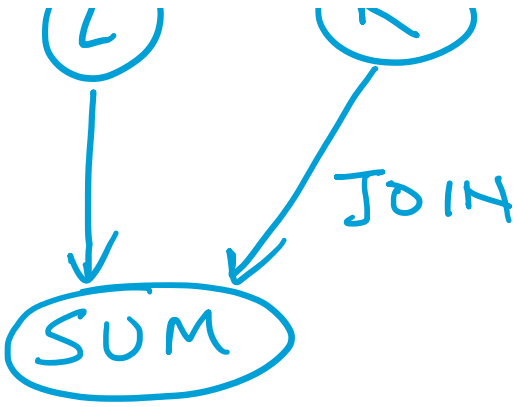


HALF}

HALF}

m2.get()

ts



Memoization

$y1 = G(x1)$ → insert(G, x1, y1)

$y2 = G(x2)$

$y3 = G(x8)$ — normal sequ

LOOKUP(G, x1) prog

$F\ y1 = F\{G(x_1)\}$ → Insert (G, x

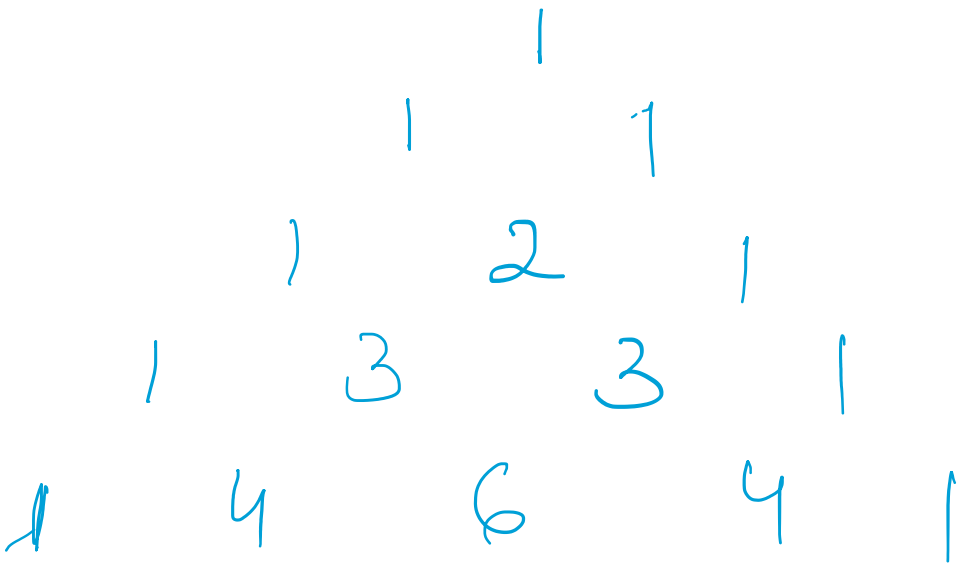
$y2 = G(x2)$

$y3 = LOOKUP(G, x1).get()$

Example : Pascal's triangle

ential
em

1, F41)



So in each case, over here, what you have is an element is the sum of two neighboring elements in the previous row. This is a well-known technique for computing the binomial coefficients. Now, this lends itself to parallel functional programming very nicely because you can imagine a recursive divide and conquer approach where tasks to compute 4 and 6 will recursively create tasks to compute the coefficients in the previous rows. But we want to ensure that we don't compute this value 3 two times when it's used by 4 and by 6. So the way to do this is, again, by memoization. And if you use the same schema that we had over here-- where we are inserting futures for sub-computations over here and doing a get operation for the return values-- then we get a data structure that captures the futures. And we ensure that we don't compute this value in the previous rows multiple times. It's only computed once. In fact, one very interesting property over here is that if you have parallel tasks to compute these two coefficients, 4 and 6, they may try to launch a task to compute 3 in parallel. And we don't know which one will launch the task to compute this value, 3. It could be either, whichever one does it launches the future. And the other one will look up the future and then block on the task waiting on the get operation.

Java Streams

```
for (s: students)
  print(s)
```

⇒

```
students.stream().
  (s → print(s)
```

Q. Find average of all active students

Java

```
For (s: students)
```

old

```
if (s.isActive)
    activeList.add(s);
for (a : active) {
    ageSum += a.age;
}
avg = ageSum / active.size();
```

Java 8

```
students.stream().filter(s -> s.isactive())
    .map(s -> s.age).average()
```

any stream can be easily parallised by using parallel stream instead of stream

Data Races and Determinism

functional determinism is the property that the same input will always lead to the same output for a function. There's another kind of determinism that also can be valuable. And that's called structural determinism.

The idea behind structural determinism is every time you run the parallel program with the same input, you'll get the same computation graph, which refers to the structure of the parallel program.

Functional Determinism	Structural Determinism
same input	same input
⇓	⇓
same output	same computational

As we'll see, we're mostly interested in parallel programs that are both functionally and structurally deterministic. But it is possible to create parallel programs that satisfy neither of these properties or only one of these properties.
Let's take an example where we can have non-determinism, and see what can cause non-determinism. So we'll go back to our two-way parallel sum example. And let's say we have an async to compute in SUM1 the sum of the left half of the array. And then in parallel with the async, we compute SUM2, which is the sum of the upper half of the array. And then we combine them to get SUM equals SUM1 plus SUM2.
Now, you'll easily see that there's a bug in the schema that I showed just now because there is no finish statement. So what this means is when the async is computing SUM1, the parent task may be reading SUM1 in parallel. This is a very pernicious bug in parallel programming that's referred to as a "data race".

It's called a data race because there's a shared location, SUM1. And one parallel task, the async over here, is updating it. And the parent task, in parallel, could be reading it. And data races occur when you can have either a read and a write operation occurring in parallel, or a write and a write. The problem with a read and a write of the same location running in parallel is the value of the read depends on the race. Do you see the old value of the variable or the new value after the write? The reason why two writes pose a problem in a data race is that the final value depends on the order in which the writes occur. And if it's running in parallel, they could occur in either order. Two reads running in parallel are fine because you can redo the reads in either order and you'll get the same value.

Now, one very, very interesting property relating data races to determinism is what we call data-race freedom, that's the absence of data races, implies both functional determinism and structural determinism.

Data races are not a bug always. Infact in some cases people are interested in such a case. This is referred as benign non-determinism. This is the case when you run your program you get different outputs but all outputs are acceptable. For example: response to a web search query,