Summary: Functional Programming

08 August 2020 01:07

Futures: Tasks with Return Values

Future tasks are tasks with return values, and a future object is a "handle" for accessing a task's return value. There are two key operations that can be performed on a future object, *A*:

1. *Assignment* — *A* can be assigned a reference to a future object returned by a task of the f *future* { ⟨ *task-with-return-value* ⟩ } (using pseudocode notation). The content of the futur object is constrained to be *single assignment* (similar to a final variable in Java), and canno modified after the future task has returned.
2. *Blocking read* — the operation, *A.get()*, waits until the task associated with future object A completed, and then propagates the task's return value as the value returned by *A.get()*. A statement, S, executed after A.get() can be assured that the task associated with future ob A must have completed before S starts execution.

These operations are carefully defined to avoid the possibility of a race condition on a task's retu value, which is why futures are well suited for functional parallelism.

Creating Future Tasks in Java's Fork/Join Framework

Some key differences between future tasks and regular tasks in the FJ framework are as follows:

1. A future task extends the RecursiveTask class in the FJ framework, instead of RecursiveAction in regular tasks.

2. The `compute()` method of a future task must have a non-void return type, whereas it has a void return type for regular tasks.

3. A method call like `left.join()` waits for the task referred to by object `left` in both cas but also provides the task's return value in the case of future tasks.

Memoization

"memoization" is to remember results of function calls *f* (*x*) as follows:

1. Create a data structure that stores the set {(*x*_11 , *y*_11 = *f* (*x*_11)), (*x*_22 , *y*_22 = *f* (*x*_22)), . . .} for each call *f* (*x*_i*i*) that returns *y*_i*i* .
2. Perform look ups in that data structure when processing calls of the form *f* (*x*′) when *x*′ eq one of the *x*_i*i* inputs for which *f* (*x*_i*i*) has already been computed.

orm,
e
ot be

A has
Any
bject

urn

as

a

es,

uals

Memoization can be especially helpful for algorithms based on dynamic programming. In the lecture, we will use Pascal's triangle as an illustrative example to motivate memoization.

The memoization pattern lends itself easily to parallelization using futures by modifying the memoized data structure to store {(x_11 , y_11 = *future*(*f* (x_11))), (x_22 , y_22 = *future*(*f* (x_22))), . . .}. The lookup operation can then be replaced by a *get()* operation on the future valu a future has already been created for the result of a given input.

Java Streams

the statement, "students.stream().forEach(s → System.out.println(s));", is a succinct way of specifying an action to be performed on each element s in the collection, *students*. An aggregate data query or data transformation can be specified by building a *stream pipeline* consisting of a *source* (typically by invoking the *.stream()* method on a data collection , a sequence of *intermediate operations* such as *map()* and *filter()*, and an optional *terminal operation* such as *forEach()* or *average()*. As an example, the following pipeline can be used to compute the ave age of all active students using Java streams:

```
students.stream()
.filter(s -> s.getStatus() == Student.ACTIVE)
.mapToInt(a -> a.getAge())
.average();
```

From the viewpoint of this course, an important benefit of using Java streams when possible is t the pipeline can be made to execute in parallel by designating the source to be a *parallel stream* i.e., by simply replacing *students.stream()* in the above code by *students.parallelStream()* or *Stream.of(students).parallel()*. This form of functional parallelism major convenience for the programmer, since they do not need to worry about explicitly allocat intermediate collections (e.g., a collection of all active students), or about ensuring that parallel accesses to data collections are properly synchronized.

Data Races and Determinism

A parallel program is said to be *functionally deterministic* if it always computes the same answer when given the same input, and *structurally deterministic* if it always computes the same computation graph, when given the same input. The presence of data races often leads to funct and/or structural nondeterminism because a parallel program with data races may exhibit differ behaviors for the same input, depending on the relative scheduling and timing of memory acces involved in a data race. In general, the absence of data races is not sufficient to guarantee determinism.

If a parallel program is *guaranteed to never exhibit a data race*, then it must be both functionally

ue, if

e

rage

hat
,

n is a
ing

ional
rent
ses

,

If a parallel program is *guaranteed to never exhibit a data race*, then it must be both functionally and structurally deterministic.

Note that the determinism property states that all data-race-free parallel programs written usin the constructs introduced in this course are guaranteed to be deterministic, but it does not impl that a program with a data race must be functionally/structurally non-deterministic. Furthermor there may be cases of “benign” nondeterminism for programs with data races in which different executions with the same input may generate different outputs, but all the outputs may be acceptable in the context of the application, e.g., search query in a search engine.