Aalto University
School of Science

Composable Structures

CCS-E4110 Concurrent Programming
Autumn 2021 (II), week 5

Vesa Hirvisalo

Week 5 in a nutshell

- Futures and promises
  - Handling incomplete computation results and concurrently computing them
- Parallel collections
  - Making sequential collections parallel
- Implementing primitives
  - Algorithms and HW support

Futures and promises

Referring to things not yet computed

- In general, a future or promise means a thing that will eventually become available
  - Also other names are used for such computation
    - But for all the basic ideas is that a handle to a value can exist before the computation of the value is completed
      - The handle be passed and forwarded (e.g., assigned somewhere)
      - Concurrently, the value is being computed
- Usually, such constructs have at least two states
  - incomplete: the computation is not yet done (i.e., wait)
  - completed: the computation is done and a value is available

Variations

- The basic idea is very generic
  - We are talking about of a form of lazy computing
    - With synchronization
- (And, of course) there are several variations
  - The way of the synchronization is the most essential
  - Some are blocking (synchronous)
    - Others are completely non-blocking (asynchronous)
- For some systems, there is only one abstraction
  - For other systems, there two abstractions; then usually
    - A future is a read-only placeholder (a stub)
    - A promise is a single assignment container that can set the value of the related future

Futures and promises in Scala

- Scala has futures and promises
  - It uses the idea of having read-only handles (references) and single assignment containers (variables)
- Futures and promises are separate but connected
  - The declarations and use is separate
  - Promise can be thought of as a writable, single-assignment container, which completes a future
  - Usually Futures are preferred over Promises as they force more encapsulation (e.g., anyone with a reference to a promise can complete it)
- A way to create asynchronous execution in Scala
  - This happens without explicitly using threads

Scala futures

- Scala futures reside in `scala.concurrent.Future`
  - You usually also need the default Execution Context
  - `scala.concurrent.ExecutionContext.Implicits.global` is to be imported
- `Future[T]` is an object holding a value of type T
  - Can hold anything (extends `Awaitable[T]`)
  - It provides an interface to read the computed value
- Futures start executing immediately when they are constructed
  - Completion can be result in `Success` or `Failure`

An example

- A callback can be registered to be called when a Future completes

```
1  import scala.util.{Success, Failure}
2
3  val f: Future[List[String]] = Future {
4    session.getRecentPosts
5  }
6
7  f onComplete {
8    case Success(posts) => for (post <- posts) println(post)
9    case Failure(t) => println("An error has occurred: " + t.getMessage)
10 }
```

Waiting

- Sometimes we might want to block the main thread until a Future has been finished.
- We can use `Await` for this:

```
val result = Await.result(myfuture, 1 second)
```

  - If the Future `myfuture` does not return 1 second, the code will throw an exception
- This blocking style of use is not common
  - The idea of concurrent execution is somewhat lost
  - Callbacks are the preferred way in Scala

Scala Promises

- For the use, the set-up is similar as Futures
  - Usually you import both `Future` and `Promise`
  - The needed execution context is the same
- `Promise[T]` is a container type for a value of type T
  - As for `Future`, can contain anything (extends `AnyRef`)
- Futures can be created by using Promises
  - Using a promise one can either successfully complete it with the `success` method, giving a Future its value
  - Alternatively a promise can be failed using the `failure` method

An example

- Using futures and promises

```
1  import scala.concurrent.{ Future, Promise }
2  import scala.concurrent.ExecutionContext.Implicits.global
3
4  val p = Promise[T]()
5  val f = p.future
6
7  val producer = Future {
8    val r = produceSomething()
9    p success r
10   continueDoingSomethingUnrelated()
11 }
```

Parallel collections

Scala Parallel Collections

- One can generate parallel collection types from sequential collections
  - Operations on parallel collection types can use all the threads available to the Scala runtime
  - The load balancing and scheduling of the parallel collections is done by the Scala runtime
  - Due to the overhead of creating new threads, better performance is only obtained for operations which are CPU heavy per item in the collection, or when the collections are quite large
- The parallelization is based on functional programming
  - The map operations performed should not have side effects
  - Side effects call for locking (performance degradation)

An example in Scala Collections

- Consider the following piece of Scala code using collections:

```
1 val list = (1 to 10000).toList
2 list.map(_ + 42)
```

  - Note: the code use *a single thread* to add 42 to each member

- When run through the Scala interpreter, we get

```
scala> val list = (1 to 10000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
scala> list.map(_ + 42)
res0: List[Int] = List(43, 44, 45, 46, 47, 48, 49, 50, 51, 52, ...
```

The example revisited

- Scala Parallel Collections
  - Using the `par` method on the list to generate a `ParVector`,

```
1 val list = (1 to 10000).toList
2 list.par.map(_ + 42)
```

- The code adds 42 to each member of the collection, but now using *several threads running in parallel*

```
scala> list.par.map(_ + 42)
res0: scala.collection.parallel.immutable.ParSeq[Int] =
ParVector(43, 44, 45, 46, 47, 48, 49, 50, 51, 52, ...
```

Creating Parallel Collections

- By using the new keyword after importing the right package

```
1 import scala.collection.parallel.immutable.ParVector
2 val pv = new ParVector[Int]
```

- By constructing a parallel collection from an existing sequential collection using the par method of the sequential collection

```
1 val pv = Vector(1,2,3,4,5,6,7,8,9).par
```

Splitters in Parallel Collections

- The architecture is based on two core abstractions:
  - *Splitter*: A way to split a parallel collection to disjoint subparts that can be operated on in parallel
  - *Combiner*: A way to combine the results of subtasks done in parallel into a final output
- The job of a splitter is to split the parallel array into a manageable partition of its elements
  - Partitions are small enough to be operated on sequentially
  - splitters are iterators (have methods like `next` and `hasNext`)

```
1  trait Splitter[T] extends Iterator[T] {
2      def split: Seq[Splitter[T]]
3  }
```

Combiners in Parallel Collections

- Each parallel collection class needs to implement its own Combiner
  - The combine method of a Combiner takes another parallel collection as a parameter, and returns a parallel collection which has the union of the elements of the two parallel collections
  - Combiners are based on Builder from the Scala sequential collections library

```
1  trait Combiner[Elem, To] extends Builder[Elem, To] {
2      def combine(other: Combiner[Elem, To]): Combiner[Elem, To]
3  }
```

Implementing primitives

Dekker's Algorithm (1/2)

boolean wantp ← false, wantq ← false
integer turn ← 1

P	Q
loop forever	loop forever
p1: non-critical section	p1: non-critical section
p2: wantp ← true	p2: wantq ← true
p3: while wantq	p3: while wantp
p4: if turn = 2	p4: if turn = 1
p5: wantp ← false	p5: wantq ← false
p6: await turn = 1	p6: await turn = 2
p7: wantp ← true	p7: wantq ← true
p8: critical section	p8: critical section
p9: turn ← 2	p9: turn ← 1
p10: wantp ← false	p10: wantq ← false

Dekker's Algorithm (2/2)

- Dekker's algorithm is the first correct solution to the CS problem
  - Guarantees mutual exclusion, freedom from deadlock and freedom from starvation for two processes
  - Freedom from starvation is irrespective of scheduling policy, as long as the weak fairness assumption holds
  - Instead of a naive turn taking, the algorithm features an alternating priority which will guarantee that one process may proceed to the critical section, if both are seeking to do so. After exiting the critical section, the priority is given to the other process
- Dekker's algorithm works with classical memory models!
  - Not with modern weak memory models such as Scala (JVM), C++11, or the x86 memory model!
- There are several other known solutions to the CS problem

Hardware support

- The CS problem can be solved easily with hardware support
  - Thus far we have assumed that our hardware does not have support for complex atomic actions
  - Modern instruction sets provide hardware level implementations of such complex actions as atomic machine instructions
  - Common instructions include: Test-and-set, Exchange, Compare-and-swap, Fetch-and-add
- Using the specifically designed HW instructions in the implementation of primitives is the good approach
  - But application programmers **seldom implement the primitives**, they typically **use the primitives** provided by the OS/compiler/etc. developers (who hopefully know their job)

Using “Test” and “Test-and-Set”

Critical section using TTS	
integer lock ← 0	
P	Q
integer local1 loop forever non-critical section p1: repeat p2: while(lock = 1) skip p2: ts(lock, local1) p3: until local1 = 0 p4: critical section p5: lock ← 0	integer local2 loop forever non-critical section p1: repeat p2: while(lock = 1) skip p2: ts(lock, local2) p3: until local2 = 0 p4: critical section p5: lock ← 0

Using Hardware Support in Practice

- The critical section code has actually two parts
  - Entry protocol (the hand-shaking for getting exclusive access)
  - Exit protocol (the hand-shaking for releasing the exclusion)
- In practice, more than critical section code is needed
  - Many of these are related to memory
    - Waiting for various memory operations/events
  - A memory fence is an typical example
    - Modern CPUs employ performance optimizations that can result in out-of-order execution
    - A fence enforces an ordering constraint on memory operations issued before and after the fence instruction

The basic needs

- Thread-safe code
  - The code can be concurrently executed
    - Note that thread-safety can be a platform-specific things, i.e., code that is thread-safe on a platform may not be such elsewhere
  - Usually (when speaking about thread-safe code), the focus is on the use of shared data (shared memory is assumed)
- Re-entrant code
  - A rather similar thing, but usually the focus is on constructing the data to be used in a shared way
- Note that in addition to data, the code or code closures* itself can be mutable objects
  - Such situations are very tricky

(*) Often a function together with a referencing environment for the non-local variables of that function, e.g., Haskell is rich with these features.

On the basic approaches

- Synchronization based approaches
  - Concurrency is based on using high-level primitives, e.g., Scala monitors or semaphores
  - Synchronization primitives often are connected to software based waiting queues
- Lock-free approaches
  - Concurrency is based on using low-level primitives, e.g., atomics or specific machine instructions*
  - Usually such primitives do not use software queues
- Data structures
  - Very often based on imbalance between reading and writing

(*) See Lecture 6 Slides for more info.

Readings for collections

http://docs.scala-lang.org/overviews/parallel-collections/overview.html

Exercises => Check A+