SCL for CSO Programmers

Gavin Lowe

April 11, 2022

This document describes SCL (Scala Concurrency Library) for those familiar with CSO (Concurrent Scala Objects).

SCL is heavily influenced by Bernard Sufrin's CSO. Most of the names of classes and functions are unchanged. However, a few changes have been made, for example to provide a simpler interface.

The emphasis in SCL is on pedagogy, rather than out-and-out performance. SCL aims to provide a fairly minimal interface, essentially just enough for the Concurrent Programming course. In particular, SCL makes much less use of factory methods than CSO.

The classes and top-level functions described below can be obtained by including the line

```
import ox.scl._
```

at the top of the file.

Computations

In SCL, a ThreadGroup represents a collection of threads. The declaration thread{ comp } creates a computation that, when run, executes comp. As with CSO, computations can be combined in parallel using ||.

SCL syntax	CSO syntax	Comments
thread{comp}	proc{comp}	A thread that, when run, executes comp.
ThreadGroup	PROC	Type of a collection of threads.
p || q	p || q	Parallel composition of p and q.
|| (ps)	|| (ps)	Parallel composition of the collection of Computations ps.
run(p) or p.run	run(p) or p()	Execute the threads p.
fork(p) or p.fork	p.fork	Executes p in a new JVM thread.

Exceptions are treated slightly differently from in CSO. When run is used, if a thread throws a non-Stopped exception, then it is caught, all the other threads in the computation are interrupted, and the exception is re-thrown. If a thread throws a Stopped exception, then all the other threads in the computation are allowed to terminate, at which point the Stopped exception is re-thrown. When fork is used, if any exception is thrown, the program exits.

Channels

SCL makes no distinction between shared and unshared ports. There are just two types of channels: synchronous (SyncChan) and buffered (BuffChan). SCL does not provide factory methods for channels, so a channel can be constructed with, for example, **new** BuffChan[A](size)[1].

As with CSO, a channel comprises an InPort and an OutPort. The syntax for standard sends and receives is unchanged. The syntax for time-bounded sends and receives is changed slightly (see below).

The syntax for closing channels is mostly unchanged, except the closeIn operation has been removed (it was equivalent to close). closeOut has been renamed to endOfStream (this operation normally signals the end of a stream). In addition, operations isClosed and reopen have been added.

SCL syntax	CSO syntax	Comments
new SyncChan[A]	OneOne[A](), OneMany[A](), ManyOne[A](), ManyMany[A](), N2N[A](m,n)	Creation of synchronous channel.
new BuffChan[A](size)	OneOneBuf[A](size), ManyManyBuf[A](size), N2NBuf[A](m,n,size)	Creation of buffered channel with capacity size.
in?()	in?()	Receive from InPort in.
out!x	out!x	Send x on OutPort out.
c.close, in.close	c.close, in.closeIn	Fully close the channel.
out.endOfStream	out.closeOut	Close the channel for sending, normally signalling the end of the stream.
c.isClosed	—	Is the channel closed?
c.reopen	—	Re-open a closed channel.
in.receiveWithin (millis)	in.readBefore(nanos)	Receive from in, or timeout after millis ms/ nanos ns, returning an Option value.
out.sendWithin (millis)(x)	out.writeBefore (nanos)(x)	Send x on out, or timeout after millis ms/nanos ns, returning a boolean value.

Alternation

The syntax for alternation (alt and serve) is largely unchanged. Unlike CSO, parentheses must *not* be placed around the guard and port. The following example (a two-place buffer) illustrates most of the syntax.

```
var x = −1; var empty = true
serve(
  !empty && out =!=> {x} ==> { empty = true }
  | empty && in =?=> { v => x = v;  empty = false }
  | !empty && in =?=> { v => out!x; x = v }
)
```

Alternations in SCL have slightly fewer restrictions than in CSO. It is possible for a port to be shared between an alt and a non-alt. A port may be simultaneously enabled in

[1]The type A of data for a BuffChan must have an associated ClassTag. When A is a parametric type parameter, it is enough to give the type bound A: scala.reflect.ClassTag. If the parameter A is subsequently instantiated with a concrete type T, it may be necessary to provide an additional argument scala.reflect.ClassTag(classOf[T]).

several branches of the same alt (all but one instance will be ignored). However, the following restrictions remain:

- A port may not be simultaneously enabled in two alts. This restriction could be removed without too much difficulty.
- A channel may not have both of its ports simultaneously enabled in alts. Such usage could lead to a deadlock.

The implementation checks the above restrictions, throwing an exception if they are broken.

As with CSO, the expressions defining the ports are evaluated *once* when a serve is created, and not subsequently re-evaluated.

Monitors/locks

JVM monitors are outside CSO/SCL, so are unchanged.

In SCL, the class Monitor is replaced by a class Lock, although each Lock supports the functionality of a CSO Monitor. A Lock can be acquired or released. A computation can be protected by a Lock using lock.mutex{comp}: this ensures comp is executed under mutual exclusion on the lock.

Conditions can be created, associated with a Lock, and used as with CSO monitors.

SCL syntax	CSO syntax	Comments
new Lock	**new** Monitor	Creation of lock or monitor.
lock.acquire	—	Acquire the lock.
lock.release	—	Release the lock.
lock.mutex{comp}	monitor.withLock{ comp}	Execute comp under mutual exclusion on the lock/monitor.
lock.newCondition	monitor.newCondition	Obtain a new condition on the lock/monitor.
cond.await	cond.await	Wait for a signal on cond.
cond.await(test)	cond.await(test)	Wait for test to become true, rechecking when a signal on cond is received.
cond.signal	cond.signal	Send a signal to a thread waiting on cond.
cond.signalAll	cond.signalAll	Send a signal to each thread waiting on cond.

Barrier synchronisations

The implementation of a barrier for n threads runs in time $O(\log n)$ (whereas the CSO implementation was $O(n)$). The downside of this is that each call to sync requires an identity parameter in the range $[0..n)$, with different threads providing different identities.

Combining barriers, unlike with CSO, do not require the starting value for the accumulation.

SCL syntax	CSO syntax	Comments
new Barrier(n)	**new** Barrier(n)	Creation of barrier object for n threads.
barrier.sync(me)	barrier.sync	Synchronisation (by thread with identity me).
new Combining-Barrier(n, f)	**new** Combining-Barrier(n, f, e)	Creation of combining barrier for n threads, with combining function f (and starting value e).
barrier.sync(me, x)	barrier.sync(x)	Synchronisation (by thread with identity me) providing input x.
new AndBarrier(n)	**new** lock.AndBarrier(n)	Creation of conjunctive combining barrier.
new OrBarrier(n)	**new** lock.OrBarrier(n)	Creation of disjunctive combining barrier.

Semaphores

Semaphores in SCL are very similar to as in CSO. An exception is that the up operation *requires* that the semaphore is in the down state (a call of up when the semaphore is already up is normally a programming error).

SCL syntax	CSO syntax	Comments
new Semaphore(isUp)	Semaphore(isUp)	Creation of semaphore with state given by isUp.
new MutexSemaphore	MutexSemaphore()	Creation of semaphore in the up state, e.g. for mutual exclusion.
new Signalling-Semaphore	SignallingSemaphore()	Creation of semaphore in the down state, e.g. for signalling.
new Counting-Semaphore(permits)	Counting-Semaphore(permits)	Counting semaphore, with permits permits available initially.
sem.up	sem.up	Raise the semaphore.
sem.down	sem.down	Lower the semaphore.

Linearizability testing

The linearizability testing framework is incorporated within SCL. (It was a separate package from CSO.) The interface has been slightly simplified from previously.

Each worker operating on the concrete datatype should have signature

```
def worker(me: Int, log: LinearizabilityLog[S, C]) = ...
```

where S is the type of the sequential specification object, and C is the type of the concurrent object being tested. Each worker performs and logs operations using commands of the form

```
log(concOp, string, seqOp)
```

where concOp: C => A is the operation performed on the concurrent object, seqOp: S => (A,S) is the corresponding operation on the specification object, and string is a String that describes the operation (with different strings for semantically different operations).

The linearizability tester is created and run using commands of the form

```
val tester = LinearizabilityTester[S,C](seqObj, concObj, p, worker _)
assert(tester() > 0)
```

where seqObj is the sequential specification object, concObj is the concurrent object being tested, p is the number of workers to run, and worker is as above.