SCL for CSO Programmers

Gavin Lowe

March 29, 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.

Philosophy.

SCL makes much less use of factory methods than CSO.

Computations

In SCL, the basic unit of a computation is a *thread* of type Thread. The declaration thread{ comp } creates a computation that when run executes comp.

A *computation* of type Computation represents the parallel composition of zero or more threads. (Thread is a subtype of Computation.) 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.
Thread	PROC	Type of a thread.
Computation	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)	run(p) or p()	Execute the threads p.
fork(p)	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 Stopped exception, then that is caught and re-thrown when the computation ends; if any other sort of exception is thrown, that halts the program immediately. When fork is used, if a thread throws any exception, that halts the program immediately.

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).

*** Not true!

In addition, a reopen operation has been added to re-open a closed channel.

SCL syntax	CSO syntax	Comments
SyncChan[A]	OneOne[A], OneMany[A], ManyOne[A], ManyMany[A], N2N[A]	Types of synchronous channels
BuffChan[A	OneOneBuf[A], ManyManyBuf[A], N2NBuf[A]	Types of buffered channels.
c?()	c?()	Receive from InPort c.
c!x	c!x	Send x on OutPort c.
c.close	c.close, c.closeIn	Fully close the channel.
c.closeOut	c.closeOut	Close the channel for sending.
c.reopen	—	Reopen the channel.
c.receiveWithin (millis)	c.readBefore(nanos)	Receive from c, or timeout after millis ms/nanos ns, returning an Option value.
c.sendWithin (millis)(x)	c.writeBefore (nanos)(x)	Send x on c, or timeout after millis ms/nanos ns, returning a boolean value.

Alternation

The syntax for alternation (alt and serve) is largely unchanged. Parentheses should 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(
```

[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.

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

- A port cannot be simultaneously enabled in two alts. This restriction could be removed without too much difficulty.
- A channel may not have both of its port used simultaneously in alts. This restriction is fairly necessary for synchronous channels, but, I think, unnecessary for buffered channels.

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

Monitors

Other

Barrier synchronisations

Semaphores.

Linearizability testing