Concurrent Programming

Gavin Lowe and Bernard Sufrin

Hilary Term 2016-17

6: Alternation

The need for alternation

- ⋆ At present, we can write processes that can commit to inputting from a single channel.

- ⋆ But it's often useful to be able to input from the first available channel of a collection of channels: the **`alt`** construct supports this.

- ⋆ It is similer to the CSP external-choice operator $\Box$.

alt – simple syntax

⋆ The command

```
alt( inport₁   =?=> {bv₁   =>  cmd₁ }
   | ...
   | inport_n =?=> {bv_n  =>  cmd_n}
   )
```

waits until one of the ports $\texttt{inport}_i$ is *ready to communicate*, and then reads a value, v from that port and applies the corresponding function $\{\mathsf{bv}_i \Rightarrow \mathsf{cmd}_i\}$ to v.

⋆ If several ports are ready simultaneously, the choice between them is made nondeterministically.

⋆ If all the ports are closed (or become closed while waiting) then an `Abort` exception is thrown.

⋆ The `inport_i =?=>{bv_i =>cmd_i}` constructs are called *input events* in CSO.

(Events in CSP are distinct from these, and shouldn't be confused with them)

Example: a simple tagger

⋆ `tagger` repeatedly inputs from one of two input ports, tags the value input, and outputs it.

```
def tagger[T](l: ?[T], r: ?[T], out: ![(Int, T)]): PROC = proc {
    repeat {
      alt ( l =?=> { vl => out!(0, vl) }
          | r =?=> { vr => out!(1, vr) }
          )
    }
    l.closeIn; r.closeIn; out.closeOut
}
```

⋆ Exercise: design a corresponding de-tagger, and consider how the two might be used together to share the bandwidth of a single channel.

Events with Boolean Guards

⋆ It's often useful to specify that a particular inport should be considered for input only if some condition, or *guard*, is true.

⋆ The following command does this:

```
alt( (guard_1  && inport_1 ) =?=> {bv_1  => cmd_1 }
   | ...
   | (guard_n && inport_n) =?=> {bv_n => cmd_n}
   )
```

- Guards are evaluated *at most once*, and *must not* have side-effects.
- An open inport with a true guard is called *feasible*.
- An event corresponding to an *feasible* inport that is ready to be read is called *ready*.

⋆ The command waits until either some events are ready, or all inports have closed.

- In the former case it executes exactly one of the ready events
- In the latter case it `Abort`s

Case study: almost-fair tagger

⋆ This `tagger` is *intended* to prevent either side from getting too far ahead.

```
def tagger[T](l: ?[T], r: ?[T], out: ![(Int, T)]) = proc {
  var diff = 0
  repeat {
    alt ( (diff< 5  && l) =?=> { lv => out!(0, lv); diff+=1 }
        | (diff> -5 && r) =?=> { rv => out!(1, rv); diff-=1 }
        )
  }
  l.closeIn; r.closeIn; out.closeOut
}
```

⋆ What happens if $diff \not< 5$ and r closes?

- Both events are infeasible
- The alt aborts and the loop terminates
- But `l` may still have some remaining input!

Rebuilding the almost-fair tagger

⋆ We tried to do everything with the original guards, but were defeated by the race condition

⋆ This two-phase solution seems least inscrutable

```
// terminate when:  diff>-5 and r closed || diff<5 and l closed
repeat
{  alt ( ((diff<  5) && l) =?=> { lv => out!(0, lv); diff+=1 }
       | ((diff> -5) && r) =?=> { rv => out!(1, rv); diff-=1 }
       )
}
// at most one of these will read more than 0 times
repeat { out!(0, l?) }
repeat { out!(1, r?) }
```

serve vs. repeated alt

⋆ Using an **alt** inside a **repeat** is very common, and there is a special form that optimises this pattern. The two constructs below behave similarly

```
serve( (g_1 && p_1 ) =?=> {v_1 => c_1 } |····| (g_n && p_n) =?=> {v_n=>c_n} )

repeat {
  alt( (g_1 && p_1 ) =?=> {v_1 => c_1 } |····| (g_n && p_n) =?=> {v_n=>c_n} )
}
```

except that

- the former creates a single **alt** object which is used repeatedly; and *approximates* fairness using a "round-robin" policy of choosing between ports that are simultaneously ready in successive iterations.

  For example, if all of the p_i are ready on $> n$ successive iterations, then they will be chosen in the order $p_1, p_2, ..., p_n, p_1, p_2, ...$.
- the latter reconstructs an **alt** object on each iteration, and has no knowledge of what happened on the last iteration, so cannot be fair.

Here's yet another tagger.

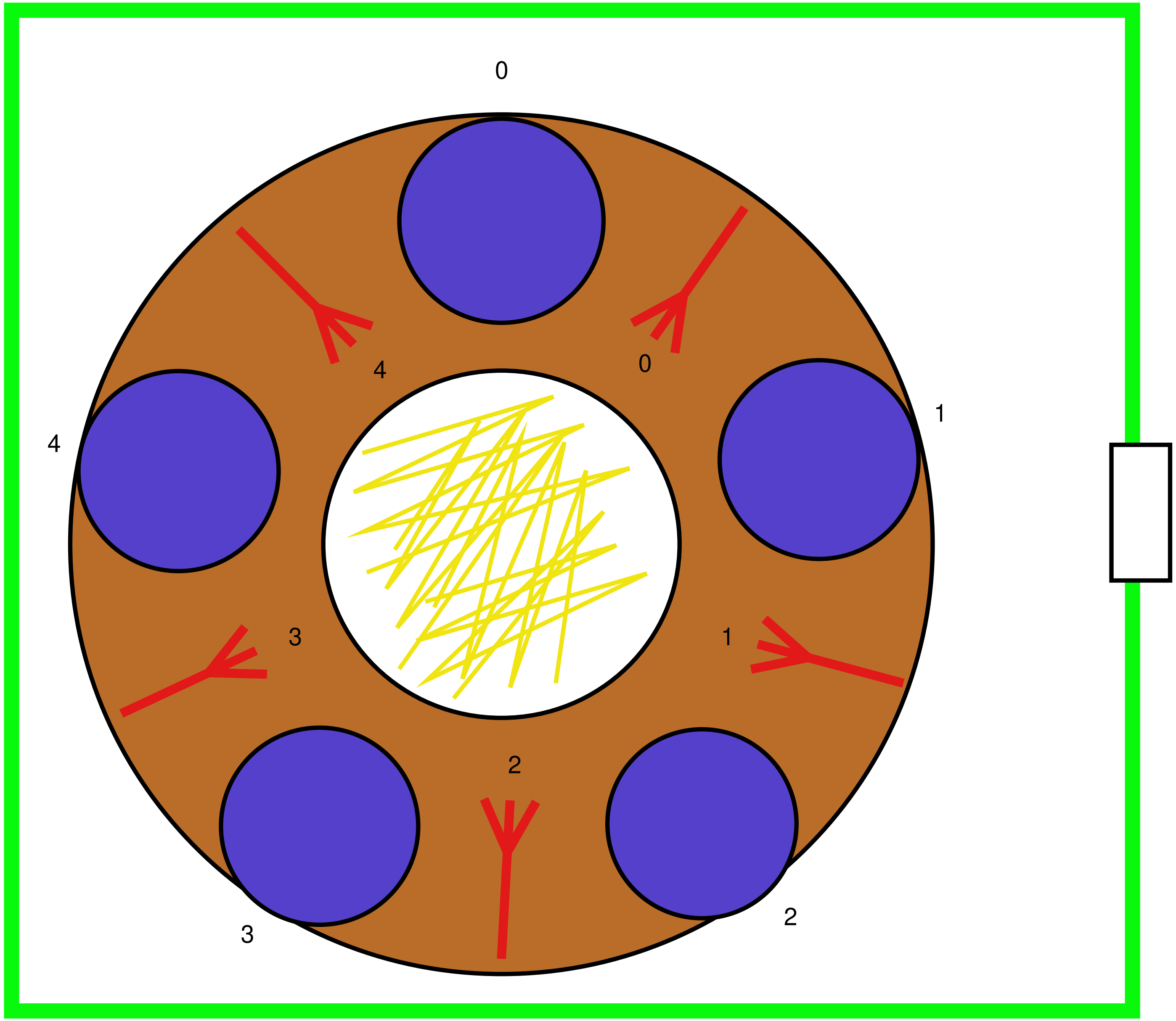
```
def tagger[T](l: ?[T], r: ?[T], out: ![(Int, T)]) = proc {
    serve( l =?=> { lv=>out!(0, lv) }
         | r =?=> { rv=>out!(1, rv) }
         )
    l.closeIn; r.closeIn; out.closeOut
}
```

Exercise: how (and under what circumstances) will it behave differently to the earlier one?

The Dining Philosophers – a parable of deadlock avoidance

The story:

- ⋆ 5 philosphers spend their lives thinking and eating.
- ⋆ They share a common dining room, which has a circular table with 5 chairs around it, a plate in front of each chair, and a big bowl of spaghetti in the middle.
- ⋆ There are 5 forks – placed between the 5 plates at the table.
- ⋆ After thinking for a while a philosopher gets hungry, picks up the fork to her left as soon as it's available, then picks up the fork to her right as soon as it's available.
- ⋆ Once she has two forks, she serves herself and spends some time eating.
- ⋆ Then she puts the forks down and does some more thinking.


0
4
0
1
4
3
1
2
3
2

- $\star$ If all five philosophers get hungry at about the same time and pick up their left fork, then they all starve! (Why?)

- $\star$ How can we simulate this?

- $\star$ How can we solve the problem?

Dining Philosophers: towards a simulation

```
import io.threadcso._

object Phils
{
   val N = 5 // Number of philosophers

   val random = new scala.util.Random

   // Simulate basic actions
   def Eat   = sleep(500*milliSec)
   def Think = sleep(random.nextInt(800)*milliSec)
   def Pause = sleep(500*milliSec)
```

⋆ In order to see what's happening, we'll arrange for processes to send messages on the shared `report` channel, which we'll buffer.

```
val report = N2NBuf[String](size=20, writers=0, readers=1, "report")
```

⋆ We will print them with the process

```
io·threadcso·component·console(report)
```

⋆ We will give each philosopher channels to her left and right forks.

⋆ She will send messages to her forks that tells the fork who she is and what she wants to do with them:

```
abstract class Action {}
case class Pick(who: Int)  extends Action
case class Drop(who: Int)  extends Action
```

A philosopher

```
def Phil(me: Int, left: ![Action], right: ![Action]) = proc("Phil"+me)
{
  repeat {
      report!(s"$me␣sits")
      Think
      left!Pick(me);  report!(me+"␣picks␣up␣left␣fork");  Pause
      right!Pick(me); report!(me+"␣picks␣up␣right␣fork"); Pause
      report ! (me+"␣eats"); Eat
      left!Drop(me);  report!(me+"␣drops␣left␣fork"); Pause
      right!Drop(me); report!(me+"␣drops␣right␣fork"); Pause
      report!(s"$me␣gets␣up"); Pause
  }
  println(s"Phil␣$me␣DIED")
}
```

A fork

```
def Fork(me: Int, left: ?[Action], right: ?[Action]) = proc("Fork"+me) {
  var owner: String="?"
  withDebuggerFormat (s"Fork␣${me}␣with␣phil␣$owner") // register with debugger
  {
    serve
    {( left  =?=>
       { case Pick(x) =>
                owner=s"$x"
                left?()  match { case Drop(y)=>assert(y==x); owner="?"} }
       |
       right =?=>
       { case Pick(x) =>
                owner=s"$x"
                right?() match { case Drop(y)=>assert(y==x); owner="?"} }
    )}
  }
}
```

- ⋆ A fork is initially ready to be be picked by its left or its right
- ⋆ Thereafter it must be dropped from that side before it is ready to be picked again
- ⋆ We keep track of its owner and register its state with the debugger

Channels

⋆ We define the channels from the philosophers to their adjacent forks by:

```
val philToLeftFork  =
    for (i←0 until N) yield
        OneOne[Action](s"Phil($i)␣to␣Fork($i)")

val philToRightFork =
    for (i←0 until N) yield
        OneOne[Action] (s"Phil($i)␣to␣Fork(${(N+i-1)%N})")
```

◦ `toLeft(i)` is from `Phil(i)` to `Fork(i)` and we name it accordingly.

◦ `toRight(i)` is from `Phil(i)` to `Fork((i-1)%N)` and we name it accordingly.

Putting it together

```
val AllPhils: PROC =
    || (for (i ← 0 until N) yield
            Phil( i, philToLeftFork(i), philToRightFork(i) ))

val AllForks: PROC =
  || (for (i ← 0 until N) yield
          Fork( i, philToRightFork((i+1)%N), philToLeftFork(i) ))

val System =
    AllPhils || AllForks || component.console(report)
```

⋆ Notice the use of the prefix `||` construct

⋆ We are going to use the debugger, so we activate it and print its port number

```
def main(args : Array[String]) = { println(debugger); System() }

}
```

Simulation results

- ⋆ When we run this it sometimes deadlocks almost immediately; and sometimes runs for a long time without deadlocking.
- ⋆ A typical quickly-deadlocking run on Unix output the following:

```
675 $ xso Phils
Debugger(http://localhost:8000)
1 sits
2 sits
4 sits
3 sits
0 sits
4 picks up left fork
0 picks up left fork
1 picks up left fork
3 picks up left fork
2 picks up left fork
```

⋆ We point a browser at `localhost:8000` to examine the deadlocked threads. Typically:

```
THREAD Fork0#10 WAITING FOR CHANNEL Phil(0) to Fork(0): OneOne ? from Fork0#10
io.threadcso.channel.OneOne.?(OneOne.scala:127)
io.threadcso.channel.OneOne.?(OneOne.scala:172)
Phils$.$anonfun$Fork$4(Phils.scala:50)
Phils$.$anonfun$Fork$4$adapted(Phils.scala:50)
Phils$$$Lambda$143/1547338188.apply(Unknown Source)
io.threadcso.alternation.event.package$InPortEvent.run(package.scala:221)
io.threadcso.alternation.Run.findFairlyAndRun(Run.scala:341)
io.threadcso.alternation.Run.$anonfun$serve$1(Run.scala:507)
io.threadcso.alternation.Run$$Lambda$149/670152325.apply$mcV$sp(Unknown Source)
io.threadcso.package$.repeat(package.scala:322)
io.threadcso.alternation.Run.serve(Run.scala:493)
io.threadcso.package$.serve(package.scala:238)
Phils$.$anonfun$Fork$1(Phils.scala:48)
Phils$$$Lambda$121/88558700.apply$mcV$sp(Unknown Source)
io.threadcso.process.Process$Simple.apply$mcV$sp(Process.scala:55)
io.threadcso.process.Process$Handle.run(Process.scala:168)
io.threadcso.process.Process$Par.apply$mcV$sp(Process.scala:89)
io.threadcso.process.Process$Handle.run(Process.scala:168)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
java.lang.Thread.run(Thread.java:748)
```

Here we elide the details of the other thread backtraces, showing just what each is waiting on:

```
THREAD console#11 WAITING
       FOR CHANNEL report: N2NBuf  (writers=0, readers=1) size=20, length=0, remainingCapacity=20)

THREAD Fork1#12 WAITING FOR CHANNEL Phil(1) to Fork(1): OneOne ? from Fork1#12

THREAD Fork2#13 WAITING FOR CHANNEL Phil(2) to Fork(2): OneOne ? from Fork2#13

THREAD Phil1#14 WAITING FOR CHANNEL Phil(1) to Fork(0): OneOne !(Pick(1)) from Phil1#14

THREAD Fork3#15 WAITING FOR CHANNEL Phil(3) to Fork(3): OneOne ? from Fork3#15

THREAD Phil2#16 WAITING FOR CHANNEL Phil(2) to Fork(1): OneOne !(Pick(2)) from Phil2#16

THREAD Phil3#17 WAITING FOR CHANNEL Phil(3) to Fork(2): OneOne !(Pick(3)) from Phil3#17

THREAD Fork4#18 WAITING FOR CHANNEL Phil(4) to Fork(4): OneOne ? from Fork4#18

THREAD Phil4#19 WAITING FOR CHANNEL Phil(4) to Fork(3): OneOne !(Pick(4)) from Phil4#19

THREAD Phil0#1 WAITING FOR CHANNEL Phil(0) to Fork(4): OneOne !(Pick(0)) from Phil0#1
```

The debugger also prints the states of the objects registered with it.

```
CHANNEL Phil(0) to Fork(0): OneOne ? from Fork0#10
CHANNEL Phil(0) to Fork(4): OneOne !(Pick(0)) from Phil0#1
CHANNEL Phil(1) to Fork(0): OneOne !(Pick(1)) from Phil1#14
CHANNEL Phil(1) to Fork(1): OneOne ? from Fork1#12
CHANNEL Phil(2) to Fork(1): OneOne !(Pick(2)) from Phil2#16
CHANNEL Phil(2) to Fork(2): OneOne ? from Fork2#13
CHANNEL Phil(3) to Fork(2): OneOne !(Pick(3)) from Phil3#17
CHANNEL Phil(3) to Fork(3): OneOne ? from Fork3#15
CHANNEL Phil(4) to Fork(3): OneOne !(Pick(4)) from Phil4#19
CHANNEL Phil(4) to Fork(4): OneOne ? from Fork4#18
CHANNEL report: N2NBuf  (writers=0, readers=1) size=20, length=0, remainingCapacity=20)

Fork 0 is with phil 0
Fork 1 is with phil 1
Fork 2 is with phil 2
Fork 3 is with phil 3
Fork 4 is with phil 4
```

We can use the channel and fork state information to construct the cyclic dependency graph that explains the deadlock. Here we've sorted this information to make the task easier.

alt vs. ManyOne / N2N channels

- ⋆ Sometimes a `ManyOne` or `N2N(readers=1,writers=0)` channel can produce the same effect as an **`alt`**.

- ⋆ For example, consider a variant of the dining philosophers where the philosophers use channels `pick` and `drop`, and where each fork has single `pick` and `drop` channels, on which it can receive messages from either of the adjacent philosophers:

```
def Fork(me : Int, pick: ?[Unit], drop: ?[Unit]) =  proc("Fork"+me)
{
    repeat{ pick?(); drop?() }
}
```

- ⋆ Note that the `pick` communication could be from either neighbouring philosopher.

- ⋆ Why do we need separate `pick` and `drop` channels?

alt semantics

⋆ Consider an **alt** construct of the form:

```
alt( (guard₁  && port₁ ) =?=> fn₁
   | ...
   | (guard_n && port_n) =?=> fn_n
   )
```

⋆ Each guard is evaluated *at most once*.

⋆ An event is said to be

- *feasible* if its guard evaluates to true and its port is open.
- *ready* if it is feasible and its port is ready to be read.

⋆ If some events are feasible the construct waits until at least one of them is ready, and selects one of the ready events to execute.

⋆ If no event is feasible (or all events become infeasible during the wait because their ports close) an `Abort` exception (a subclass of `Stop`) is raised.

serve (and repeated alt) termination

⋆ Recall that a **serve** construct of the form:

```
serve ( (guard₁  && port₁ ) =?=> fn₁
      | ...
      | (guard_n && port_n) =?=> fn_n
      )
```

is effectively a "fair" form of

```
repeat{ alt ( (guard₁  && port₁ ) =?=> fn₁
            | ...
            | (guard_n && port_n) =?=> fn_n
            )
      }
```

So

- Each guard is evaluated at most once *per iteration*.
- When all events become infeasible the `Abort` exception from the **alt** is caught and the **serve** iteration terminates normally.

alt with outport guards

⋆ It is also possible to use an output port within an **alt**:

```
(guard && outport) =!=> { expression}
```

⋆ In this case, when the outport is ready to communicate the expression is evaluated and its value written to the outport.

⋆ Sometimes it's necessary to do something else *after* the value been written.

```
(guard && outport) =!=> { expression } ==> { command }
```

⋆ If it's *really necessary* to do something else with the value after it has been written, use the (ugly) circumlocution:

```
var save: T = ...
alt (
    ...
  | (guard && outport) =!=> { save=expression; save } ==> { command }
  )
```

Example: tee revisited

```
def tee[T](in: ?[T], out1: ![T], out2: ![T]) = proc {
    var v: T = in·nothing
    serve(   out1 =!⇒ { v=in?; v } ⟹ { out2!v }
         |   out2 =!⇒ { v=in?; v } ⟹ { out1!v }
         )
    in·closeIn; out1·closeOut; out2·closeOut
}
```

⋆ This can output to `out1` and `out2` in either order.

⋆ This `tee` can be more efficient than the `{out1!v}||{out2!v}` solution (when context switching and/or acquiring short-lived threads repeatedly is expensive)

⋆ Exercise: can you generalize it to a variadic output `tee[T](in: ?[T], outs: seq[![T]])`?

⋆ Notice that the variable `v` is shared in a disciplined way, because only one of the events can be operative in each cycle of the serve.

Mixing inport and outport events: a two place buffer

- ⋆ A two-place buffer process can be specified in CSP as:

$$\begin{aligned} BUFF2E &= in?x \rightarrow BUFF2NonE(x) \\ BUFF2NonE(x) &= out!x \rightarrow BUFF2E \; \square \; in?y \rightarrow out!x \rightarrow BUFF2NonE(y) \end{aligned}$$

- ⋆ In its empty state $BUFF2E$ it can only input a value x from in.
- ⋆ It then enters its nonempty state $BUFF2NonE(x)$ holding x and can either:
  1. Output x to out and enter its empty state again, or
  2. Input a second value, y, from in, after which it holds 2 values, and can then only Output x to out, then enter its nonempty state $BUFF2NonE(y)$ holding y
- ⋆ The choice between 1 and 2 is not made by the buffer itself, but by the environment in which the buffer is embedded. It is an *external* choice.

⋆ Inport and outport events can be mixed within an alternation.

⋆ The buffer can be straightforwardly implemented in CSO, discriminating between empty and nonempty states with a boolean.

```
def Buff2Alt[T](in: ?[T], out: ![T]) = proc {
  var x     = in?()
  var empty = false
  serve ( (empty  && in)  =?=> { y => x=y; empty=false }
        | (!empty && in)  =?=> { y => out!x; x=y }
        | (!empty && out) =!=> { empty=true; x }
        )
}
```

⋆ When empty the next action *must* be to input.

⋆ When !empty, either:

1. x can be output, and the buffer become empty
2. an input can be accepted, but then x must be output

Alternation vs. Monitors/Semaphores

⋆ Alternation-based solutions to complex synchronization problems are often *much* easier to understand than monitor/semaphore solutions.

⋆ Example: processes `P1, P2` must share a monotonically-increasing counter; but `P1` may increment it no more than half as often as `P2` and `P2` may only access it when it is even.

An alternation-based solution

```
def Share(inc1 : ?[Unit], inc2 : ?[Unit], getX1 : ![Int], getX2 : ![Int]) = proc
{   var c1 , c2 : Int = 0
    serve ( getX1                              =!⇒ { c1 + c2  }
          | ((c1 + c2 )%2==0 && getX2 )        =!⇒ { c1 + c2  }
          | inc2                               =?⇒ { _ ⇒  c2  += 1 }
          | (2*c1  < c2  && inc1 )             =?⇒ { _ ⇒  c1  += 1 }
          )
}
```

Exercise: implement this sharing arrangement as a monitor or with semaphores.

Restrictions on the alternation constructs

⋆ The use of **`alt`**ernations (**`alt/serve/prialt/priserve`**) is restricted by the following rules:

- An **`alt`**ernation may not have two simultaneously enabled events using the same port.
- A port (whether shared or unshared) may not simultaneously be used in an input event and non-alt read, or in an output event and a non-alt write.
- No more than one port of a channel may participate simultaneously in the execution of an alternation.

Timeouts

⋆ Sometimes we don't want an **alt** to wait for ever: we want a timeout.

⋆ In the construct:

```
alt( event_1
   | ...
   | event_n
   | after(t) ⟹ {timeoutCmd}
   )
```

the final branch acts as a timeout. If no feasible event is ready within $t$ns (where $t > 0$) then `timeoutCmd` is executed – after which the **alt** terminates.

Exercise: What should happen when there are no feasible events?

Case study: failure detection

⋆ A process A needs to know whether a peer process B has failed. This might be important when they run on different hosts or JVMs.

⋆ Solution: send regular "heartbeat" messages from B to A; if A receives no message within a particular interval, it signals an error:

```
serve( ping      =?=> { _ => () }
     | after(t)  ==>  { <signal an error> }
     )
```

⋆ B needs to send a `ping` slightly more often than every $t$ns.

⋆ Refinement: piggyback these messages on a data channel.

Dining philosophers with timeouts

⋆ It is tempting to use timeouts in the dining philosophers example, so that a philosopher detects that the second fork is not available, and so puts down her first fork, and retries later.

⋆ The timeout on picking up the second fork would be of the form

```
alt( right    =!=> { right!Pick(me) ; ··· }
   | after(t) ==>  { *** }
   )
```

But since this would break the rule about not having two ends of a channel be simultaneously in alternations, we now use the "deadlined write":

```
if (right.writeBefore(t)(Pick(me))) { ··· } else { *** }
```

⋆ Exercise (Practical 4): fill in the details.

As a symmetry breaker, we probably need to make the philosophers wait a random amount of time before re-trying.

orelse

⋆ Recall that if no event is feasible in an **alt** then an `Abort` is raised.

If an **orelse** branch is included in the **alt** then this doesn't happen:

Here if no event is feasible then `orelseCmd` is executed.

```
alt( event_1  | ··· | event_n
   | orelse ⟹ {orelseCmd}
   )
```

⋆ If an **after** and an **orelse** branch are included, then the `afterCmd` is executed if no feasible event becomes ready before the timeout; the `orelseCmd` is executed if (and when) all events are infeasible.

```
alt( event_1 | ··· | event_n
   | after(timeOut) ⟹ {afterCmd}
   | orelse         ⟹ {orelseCmd}
   )
```

prialt and priserve

- ⋆ Recall that if several events of an **`alt`** are ready, the choice between them is effectively made *nondeterministically*
  - ○ in effect, the choice mechanism is implementation-dependent
  - ○ and in CSO implementations **`serve`** (or its equivalent) uses a "round robin" policy to make successive choices so as to provide a form of fairness.
- ⋆ A **`prialt`** is like an **`alt`**, except that if several events are ready it selects the first one.
- ⋆ A **`priserve`** is like a **`serve`**, but also gives priority to earlier events.
  - ○ starvation of the later branches is a potential issue here
  - ○ carefully designed boolean guards can sometimes avoid such starvation

Generalized alt

- ⋆ **alts** (and **serves**, **prialts** and **priserves**) can be constructed from any mixture of singleton events, and collections of events.
- ⋆ For example, here is a generalization of the tagger that, whenever no input is ready, sends a reminder signal `()` to any (one) of the `sources` channels that is prepared to listen to it. It closes down if nothing happens for 5 seconds.

```
def nagger[T](sources: Seq[![Unit]], ins: Seq[?[T]], out: ![(Int, T)]) =
proc {
   priserve (
   | (for ( i←0 until ins·length) yield ins(i) =?⇒ { x ⇒ out!(i,x) })
   | (for ( chan ← sources) yield chan =!⇒ { () })
   | after(5*Sec) ⟹ stop
   )
   for (in ← ins) in·closeIn
   for (chan ← sources) chan·closeOut
   out·closeOut
}
```

Extended Rendezvous

⋆ Consider the problem of monitoring the traffic down an unbuffered (synchronized) channel

```
                    SyncChan
Producer  ----------------------------> Consumer


            SyncChan                  SyncChan
Producer  ------------- [ Probe ] ------------> Consumer
                            |
                            |
                            +-----------------> Monitor
```

Q: Can we use a Tee component to act as a Probe?

A: Not if the producer-consumer relationship depends on synchronization

(a Tee behaves like a one-place buffer between its producer and each of its consumers)

Extended Rendezvous Read

⋆ Solution makes use of the *extended rendezvous* read operation

⋆ Syntax and semantics of extended rendezvous read

When `in: ?[T]` and `f:T⇒U` the *extended rendezvous read*

```
in ?? f
```

waits (if necessary) for a corresponding `!`, and then yields the same value as

```
f(in ?)
```

But *termination* of the `!` is *suspended until the end of the computation of* `f`

⋆ The *synchronization rendezvous* between producer and consumer lasts from the producer's `!` starting to the consumer's computation finishing.

⋆ Examples

- Repeatedly read x from `in` then output x to `out` – effectively a one-place buffer from `in`'s ouput port to `out`'s input port.

```
def eg0(in: ?[T], out: ![T]) =
    repeat { in?{ (x:T) ⇒ { out!x } } }
```

- Repeatedly read x from `in` then output x to `out` – effectively a synchronous channel from `in`'s ouput port to `out`'s input port

```
def eg1(in: ?[T], out: ![T]) =
    repeat { in??{ (x:T) ⇒ { out!x } } }
```

- Repeatedly read x from `in` then print its value on the console and output x to `out`

```
def consoleprobe(in: ?[T], out: ![T]) =
    repeat { in??{ (x:T) ⇒ { Console.println(x); out!x} } }
```

- Repeatedly read x from `in` then output x concurrently to monitor, and to `out`

```
def probe(in: ?[T], out: ![T], monitor: ![T]) =
    repeat { in??{ (x:T) ⇒ { (proc { monitor!x } || proc{ out!x })()
```

⋆ These three programs should be indistinguishable (except for their console output)

```
val mid=OneOne[T]
(producer(mid) || consumer(mid))()

val left, right=OneOne[T]
(producer(left) || consumer(right) || consoleprobe(left, right))()

val in, out, mon=OneOne[T]
(  producer(left)
|| consumer(right)
|| probe(left, right, mon)
|| console(mon))()
```

⋆ The technique used in the last two programs is called “splitting” or “probing” a channel.

Alternation Notation

There are four sorts of alternation construct:

```
alt      ( body )
prialt   ( body )
serve    ( body )
priserve ( body )
```

The body of an alternation construct is a sequence of one or more simple $events$ (as described on the next page) or $eventComp$rehensions separated by `|`, possibly followed by an **`after`** event, possibly followed by an **`orelse`** event.

$$
\begin{array}{lcl}
body & ::= & events \;\; afterEvent? \;\; orelseEvent? \\
events & ::= & event \\
events & ::= & eventComp \\
events & ::= & events \mid events \\
eventComp & ::= & \mid \text{(for ... yield } event\text{)}
\end{array}
$$

Event Notation

The *event* notation describes input-guarded events, output-guarded events,

```
(bool && inport)  =?=>  { bv => command }
(bool && inport)  =??=> { bv => command }
(bool && outport) =!=>  { expression }
(bool && outport) =!=>  { expression } ==> { command }
```

- ⋆ The second form of input event uses an *extended rendezvous*. It synchronises termination of the writing `!` with termination of the evaluation of the `command`.
- ⋆ `(true && port)` can be shortened to `port`.
- ⋆ Channels can be used in place of ports.
- ⋆ In the second form of output event the `command` can (with some inconvenience) refer to the expression value (page 25 explains how).
- ⋆ All commands must be Unit-valued.

The **`after`** and **`orelse`** events are, respectively:

```
after(nanoseconds) ==>  { command }
orelse             ==>  { command }
```

Contents

Note 1: 2 ☞

NB: this notation for input events became the CSO standard in late 2014; other forms were described in earlier papers on CSO, but are no longer supported.

Note 2: 2 ☞

In fact the input event notation admits any function of the right type to the right of the ⇒. If `inport: ?[T]`, and `fn` is *any expression* of type `T⇒Unit`, then `inport =?⇒fn` is an input event.

Note 3: Boolean guards – some details 4 ☞

- `inport =?⇒···` is equivalent to `(true && inport) =?⇒···`.
- The parentheses are necessary in `(guard && inport)`.

Note 4: An incorrect attempt to fix the fair tagger 5 ☞

- Intention: each side reads when it's behind or when the other side has closed

```
def tagger[T](l: ?[T], r: ?[T], out: ![(Int, T)]) = proc {
  var diff = 0
  repeat {
   alt ( ((!r·canInput||diff< 5 ) && l) =?⇒ { lv ⇒ out!(0, lv); diff+=1 }
       | ((!l·canInput||diff> -5) && r) =?⇒ { rv ⇒ out!(1, rv); diff-=1 }
       )
  }
  l·closeIn; r·closeIn; out·closeOut
}
```

- In fact the `!port·canInput` disjuncts do nothing to avert the subtle race condition[1]

  Consider the tagger's loop:

```
 repeat {
   alt ( ((!r·canInput || diff< 5 ) && l) =?⇒ { lv ⇒ out!(0, lv); diff+=1 }
       | ((!l·canInput || diff> -5) && r) =?⇒ { rv ⇒ out!(1, rv); diff-=1 }
       )
 }
```

[1] I am grateful to Toby Cathcart Burn (M&CS Merton, 2013) for first noticing that the !*canInput* disjuncts don't achieve the desired goal.

- Suppose that at the instant this starts executing `r.canInput && diff>=5` and `l, r` are open but neither is ready.
- The top event is infeasible because its boolean guard is false.
- The bottom event is feasible because its guard is true and `r` is open.
- The implementation now waits for "something to happen" on port `r`.
  - If `r` becomes ready, then the bottom event fires, and the iteration continues
  - If `r` *closes*, then the **`alt`** throws an `Abort`, **thereby terminating the iteration**
- The loop can terminate **despite** one of its channels still being open: **this is not the intended behaviour**.

⋆ Should we change the semantics of **`alt`** to something like:

If there are still open ports when the only feasible event becomes infeasible because of a port `p` closing (while the **`alt`** is waiting for it to become ready), then re-evaluate all event feasibilities in case any events have become feasible because of the change in port status.

- In general this can happen only if there are (effectively) disjunctions equivalent to `!p.canInput` in the guards of the channels that remain open.
- This is not statically determinable, so the upper bound on determining feasibility becomes quadratic (rather than linear) in the number of events.
- Coming up with concise and tractable proof rules would be hard.

⋆ We conclude that it would be better to redesign the particular program we were working on.

Note 5: 15 ☞
We have added a fragment to the definition of `Fork` that defines the "debuggable" description of the state of the fork at any time. Although it's not in general necessary to interface processes with the debugger kernel like this, it can be very helpful when trying to diagnose deadlocks or other incorrect behaviour.

Note 6: 21 ☞
Channels are automatically registered with the debugger until they close.

Note 7: 25 ☞
The obvious construct for doing something with the value written is:

```
(guard && outport) =!=> { expression } ==> { bv => command }
```

where the function `{bv =>command}` is applied to the value after it has been written. *But unfortunately the contravariance of the `OutPort` types makes this impossible.*

Note 8: 26 ☞

Because the expression in an output event can be a composite it is tempting to be quite ambitious there. For example, here's yet another `tee`: it remembers the set of values it has transmitted and sends them to a log when it terminates.

```
def tee[T](log: ![T], in: ?[T], out1: ![T], out2: ![T]) = proc {
  val seen = ··· Set[T] ···
  serve( out1 =!⇒
            { val v = in?; seen+=v; v } ⟹ { v ⇒ out2!v }
       | out2 =!⇒
            { val v = in?; seen+=v; v } ⟹ { v ⇒ out1!v }
       )
  in·closeIn; out1·closeOut; out2·closeOut
  for (v←seen) { log!v }
  log·closeOut
}
```

Note 9: Buff2Alt coding style 28 ☞

The two input branches can be merged, and `if (empty)` used to discriminate between the two states when `in` is ready.

Note 10: Delivering the shared counter as a class 29 ☞

The Share process uses channels: it can be delivered as a conventional class by defining a class in which appropriate channels are defined privately and a server process is forked.

```
class Sharer
{ private val sync1, sync2 = OneOne[Unit]
  private val int1,  int2  = OneOne[Int]
  def getX1 = int1? ()
  def getX2 = int2? ()
  def inc1 = sync1! (())
  def inc2 = sync2! (())

  ··· Share definition

  fork(Share(sync1, sync2, int1, int2))

  def close = { sync1.close; sync2.close; int1.close; int2.close }
  override def finalize = { super.finalize; close }
}
```

Aside: Garbage collection, and freeing up resources.

Each "live" Sharer object ties up a running thread; and since threads are relatively scarce resources the question of how to liberate this thread when a Sharer object can no longer be used arises.

It is a simple matter to terminate the running server (and thereby liberate its thread) if a Sharer object gets garbage collected.

The finalizer method of the object just closes all the channels used to communicate with the server, thereby causing all the events in the **serve** loop to become infeasible, and that loop to terminate.

But tying up a Sharer's thread until the garbage collector just happens to notice that it's no longer useful may not always be appropriate; indeed there is no guarantee that the Sharer will ever be finalized in this way. So it also makes sense to have a "scope-based" way of invoking finalizations.

There follows an example of one pattern we have found useful: entities whose implementation may use scarce resources are declared ($s1, s2$ in our example), then the scope in which they will be used is embedded in a **try/finally** block that invokes their finalizers as the block terminates (normally or by exception).

```
{  val s1, s2 = new Sharer
   try
   {
    ··· scope of uses of \SCALA{s1, s2}
   }
   finally
   { s1·finalize; s2·finalize }
}
```

Note 11: Restrictions on alts experimentally removed in 2010 30 ☞

The implementation of **alt**ernations in our first CSO implementation was revised by Gavin Lowe in 2010 to be more expressive than this restriction allows: *both* ports of a channel could participate in (different) **alt**ernations simultaneously. Gavin's implementation was proven (probabilistically) correct, but it required a complex multiphase synchronization. In practice the duration of this synchronization could not easily be predicted, so the restriction was reimposed in the ThreadCSO implementation for the sake of maintaining simplicity, efficiency, and predictability.

Note 12: A (shared) port may not simultaneously be used in an alt and a non-alt 30 ☞

This restriction is because the **alt** is not responsible for performing the actual read or write; without this restriction, the **alt** could choose a channel, but the communication on the channel could be pre-empted by the non-**alt**.

Note 13: alternation timeouts 31 ☞

If all events of an alternation with a timeout are infeasible or become infeasible during the timeout because channels are closing, then the alternation aborts.

The following "experiment" should confirm this.

```
import io.threadcso._
object TimeoutAlt {
  def main (args: Array[String]): Unit =
  { val c1, c2, c3 = OneOne[Int]
    def closer(m: Int, out: ![Int]) = { out!m; sleep(seconds(2)); out.closeOut }
    def alternation = {
        alt( c1 =?=> { x => println(x) }
           | c2 =?=> { x => println(x) }
           | c3 =?=> { x => println(x) }
           | after(seconds(1)) ==> { println("after") }
           )
     }

     (  proc { closer(1, c1); closer(2, c2); closer(3, c3) }
     || proc { repeat { alternation } }
     )()
    }
  }
```

Note 14: orelse event within serve loops 34 ☞

If the **orelse** branch is taken in the body of an iterative alternation (a **serve** or **priserve**) then the iteration terminates after the command guarded by **orelse** has been executed.

Note 15: 34 ☞

An event is feasible if its guard is true and its port is open. So saying "no events are feasible" or "all events are infeasible" is the same as saying "the ports of all events with true guards are closed."

Note 16: Generalized alt guard group notation 36 ☞

The notation for generalized **alt**, *etc.* guards may appear, at first sight, to be a little delicate. Notice the parentheses around the **for** ⋯ **yield** guard group constructs and the use of the vertical bar to begin each guard and guard group.

The use of `chan!(())` rather than the more natural `chan!()` avoids provoking a silly warning message from recent Scala compilers.

Note 17: 37 ☞

It may be worth doing a number of experiments to convince yourself of this; for example by monitoring the traffic to the room in the shared-channel implementation of the Dining Philosophers. What you will probably (I haven't tried it) observe is that eventually a philosopher has her enter message accepted by the buffer, and proceeds to request a fork – "thinking" she is already in the room – before the room receives her enter message. This leads to deadlock.

Note 18: 40 ☞

The speed of communication between the producer and the consumer can be controlled, in the last of these programs, by the speed with which input is accepted by the `mon` channel. This could be used as the basis of an "animation" system that permits processes to proceed step by step at the convenience of a programmer who wishes to observe detailed behaviour.