Communicating Scala Objects

Bernard Sufrin

CPA, York, September 2008

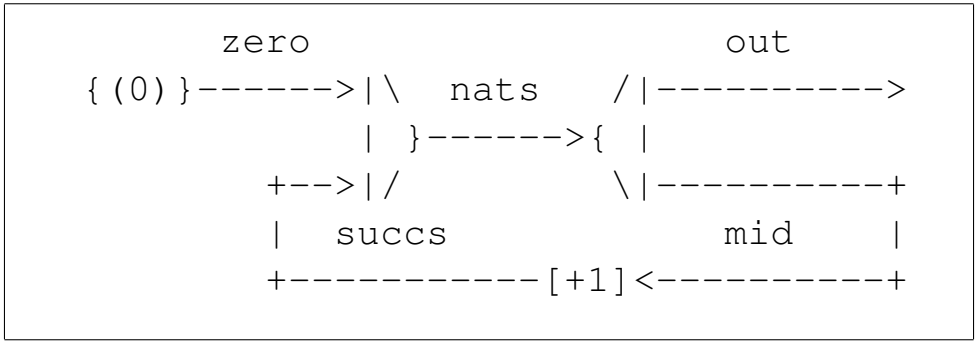
Communicating Scala Objects – why bother?

- The dogma "$Component \equiv Object$" is **no longer sustainable**
- Can "$Component \equiv Process$" be rehabilitated?
- **occam**-style (stream-oriented) programming
  - a powerful *conceptual/analytical* tool ... still!
  - increasingly becoming efficiently realizeable ... again!
  - neglected in mainstream curriculum ... somewhat!
- **occam** model not *integrated* into a mainstream language ... JCSP notwithstanding.

Scala

- Conventional object-oriented language very loosely based on (Generic) Java
- More *directly* (*i.e.* without transliteration) expressive for our purposes than Java
  - no artificial distinction between primitive and reference types
  - by-name as well as by-value parameters
  - functions/methods are first-class values
  - multiple (mixin) inheritance
  - concrete notation somewhat “plastic”
- Type system more expressive than Java’s
- “Open” compiler ... well, nearly!
- Translates to JVM and to .NET

Natural number circuit component (adapted from JCSP Plug'N'Play)




```
def nats(out: ![long]) =
{ val mid, nats, succs, zero = OneOne[long]
  (  merge (zero, succs) (nats)
  || delta (nats)        (out, mid)
  || map   { n => n+1 }  (mid, succs)
  || proc  { zero!0 }
  )
}
```

- nats(someChannel)
  - Declares four internal point-to-point synchronous channels
  - Yields a process that *when started* outputs numbers along someChannel
  - Terminates when all its components terminate ... after someChannel is closed.

```
def merge[T](l: ?[T], r: ?[T]) (out: ![T]) = proc
{
  repeat { alt( l --> { out!(l?) } | r --> { out!(r?) } ) }

  out.closeout
  l.closein
  r.closein
}
```

merge(l, r)(out) yields a process that *when started* merges l and r onto out

- ◇ It terminates when both l, r have been closed, or when out refuses
- ◇ Refusal is implemented by closing
- ◇ As written it prioritizes input from l
- ◇ This is because the **alt**'s internal data structure is rebuilt on each cycle

NB: alt notation changed since the paper; revised paper on my website

◇ Events may have boolean guards

◇ A false guard excludes its channel from the readiness scan

```
var lnum, rnum = 0
repeat { alt ( l (! r.isOpen || rnum-lnum>4) --> { out!(l?); lnum++ }
             | r (! l.isOpen || lnum-rnum>4) --> { out!(r?); rnum++ }
             )
       }
```

◇ Here neither channel can get too far ahead of the other

◇ A more general, and fair(er), merge, that outputs SOMETHING at least every 30ms

```
def merge[T](inports: ?[T]*) (out: ![T]) = proc
{ repeatalt( after(30)                          --> { out!null }
           | for (in<-inports) yield in --> { out!(in?) }
           )

  out.closeout
  for (in<-inports) in.closein
}
```

◇ An **alt** structure is built from the event collection **after ...** | **for ... yield ...**

◇ Its guard-selection method is invoked repeatedly

◇ "Fairness" implemented by "round-robin" turn-taking in the port-readiness scan

◇ **repeatalt**-loop terminates when all inports are closed or when out refuses

```
def delta[T](inport: ?[T])(outports: ![T]*) = proc
{   var buf : T = null
    val out     = || (for (port<-outports) yield proc { port!buf })

    repeat { buf = inport?; out() }

    inport.closein
    for (port<-outports) port.closeout
}
```

delta(in)(outs) yields a process that *when started* copies from in to all of outs

- ◇ out is a process that is started anew after every input
  - ... outputs v in parallel to each outport
  - ... terminates when all the outports have either accepted v or refused it
- ◇ Copying stops when inport refuses, or when any outport refuses
- ◇ Refusal is implemented by closing

◇ (Distributed) termination: the building blocks

- Closing a channel: “Henceforth nobody will communicate *from either end*”
- Closing a port: “I will never communicate via your channel *from this end*”
- For “point-to-point” (OneOne) channels these mean the same thing

- An attempt to communicate via a closed channel throws a (form of) **Stop**
  ... in the communicating (not the closing) peer
  ... even if the communication (! or ?) has started in the communicating peer

- An **alt** whose input ports are (or become) closed throws a (form of) **Stop**

- An (uncaught) **Stop** means “terminate the closest **repeat**”

- Convention: components close their communication ports on termination

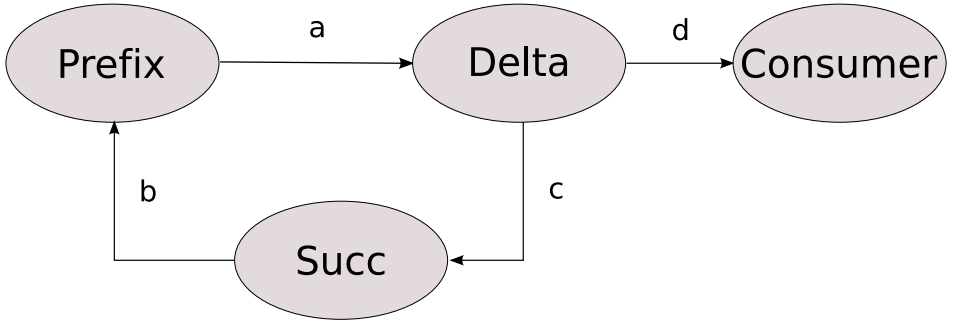
◇ Parallel compositions must “do the right thing” with uncaught exceptions

- ○ Principle: no component should fail silently, save by design
- ○ Consequence: an (uncaught) exception must cause termination
- ○ Consequence: an (uncaught) exception must be propagated from a composition

- ○ A parallel composition terminates when all its components have terminated
  - · If all components terminate without an exception,
    the composition just terminates
  - · If any components terminate with an exception,
    the composition terminates by re-throwing the ∪ of all the exceptions

Stop ∪ **Stop** = **Stop**
Stop ∪ **non-Stop** = **non-Stop**
non-Stop ∪ **Stop** = **non-Stop**
non-Stop ∪ **non-Stop** = **non-Stop**

◇ Synchronous channels performance: Actors, CSO and JCSP (Avg. μs per cycle)



ParDelta: `val v=a?;(c!v||d!v)()` SeqDelta: `val v=a?;c!v;d!v`

Host	JVM	Actors	CSO Seq	JCSP Seq	CSO Par	JCSP Par
4×2.66GHz Xeon, OS/X 10.4	1.5	28-32	31-34	44-45	59-66	54-56
2×2.4GHz Athlon 64X2 Linux	1.6	25-32	26-39	32-41	24-46	27-46
1×1.83GHz Core Duo, OS/X	1.5	62-71	64-66	66-69	90-94	80-89
1×1.4GHz Centrino, Linux	1.6	42-46	30-31	28-32	49-58	36-40

◇ Buffered Channels Actors vs CSO (Range of Avg. μs per cycle)

Host	JVM	Actors	CSO
4×2.66 GHz Xeon, OS/X 10.4	1.5	10-13	10-14
2×2.4 GHz Athlon 64X2 Linux	1.6	16-27	4-11
1×1.83 GHz Core Duo, OS/X 10.4	1.5	27-32	14-21
1×1.4 Ghz Centrino, Linux	1.6	42-45	17-19

◇ Inter-JVM round-trip medium-size message time on a TCP connection:

"no more than 20% more than ping-time on a LAN/WAN"

- CSO Under the hood
  - A process is a template for a thread
  - When a process is started a thread is acquired for it (from a self-expanding pool)
  - When processes terminate, their threads are returned to the pool
  - Pool is parameterized by keep-alive time for “resting” threads
  - Communication implementation straightforward

- Actors under the hood
  - Not *necessarily* a 1-1 correspondence between Actors and Threads
  - Actions following message receipt *may* be performed in sending Thread
  - Efficient scaleability depends on programming style!
  - Cyclic networks of actors are not efficiently scaleable.
  - Communication implementation “ingenious”

Challenges

- Technical
  - Liberate CSO from the status of tutorial toy – construct a useful component base
- Pedagogical
  - (Re)habilitate the ideas of **occam** within *mainstream* programming
- Technical (for everyone who cares)
  - Liberate threads from the awful weight of the “standard model”
  - Make local communication as efficient as method call in JVM/.NET/OWHY

Note 1 (Page 1)
It is no longer possible to preserve the illusion that OOP is “mostly” about assembling sequential programs to run in “mostly” sequential contexts.

Note 2 (Page 1)
The widespread acceptance of “design by contract” may have served to prolong its lifespan somewhat, but the overall weakness of the idea of a contract imposes serious limitations on compositionality.

I am interested in rehabilitating the idea of Component as Communicating Process

Note 3 (Page 1)
Just what integration into a mainstream language might be would be the subject of another conference!

occam itself occupied an interesting ecological niche at an interesting moment in the evolution of computing technology. The restrictions on expressive power that led to its being very efficiently implementable with little or no “kernel” support didn’t constitute an enormous step back from too many of the languages then widely accepted in the mainstream.

I am very excited at the prospect of seeing a functioning operating system kernel written in one of the modernized dialects of **occam**.

I fear that I am very ignorant of the progress that the pi calculus and join calculus have been making into the mainstream. I am aware of a number of toy languages and libraries, and of Polyphonic $C^\sharp$ but have not explored their potential.

Note 4 (Page 2)

whose class libraries may be exploited

Note 5 (Page 3)

- ◇ Formal parameter of nat is an integer output *port*
- ◇ Actual parameter of nat (likely to be) an integer *channel*
- ◇ A OneOne[T] is a Chan[T], so its type is a subtype of both ![T] and ?[T]