Lecture 18
Concurrent Programming

31st October 2002

CCS Continued

In today's notes I am going to use a and a' to stand for a label and the corresponding co-label rather than the overbar notation used last time.

I left out one important agent last time: 0. 0 is the agent that does nothing. At the end of an agent expression it means essentially *stop*.

Example agents:

$C_1 \equiv in'.out.0$
$C_2 \equiv in'.out.C_2$
$D_1 \equiv in'.outa.outb.D_1$
$D_2 \equiv in'.(outa.0 + outb.0)$
$D_3 \equiv in'.(outa.D_3 + outb.D_3)$
$D_4 \equiv in'.outa.D_4 + in'.outb.D_4$

Understand how each of these agents evolves as communication occurs.

Note well the difference in behavior between D_3 and D_4. The former receives a message on in then offers a choice to the environment of either $outa$ or $outb$. The latter, after accepting a message from in will be prepared to send only one of $outa$ or $outb$. $D_3$is like:

```
recv(in);
select[(outa, outb)]
```

while $D_4$is like:

```
choice = random boolean value;
if choice {
   recv(in); send(outa);
```

```
else {
   recv(in); send(outb);
}
```

The important point here: D_3 leaves the choice of whether *outa* or *outb* happens next to the environment (other concurrent processes), while D_4 makes an *internal choice.*

Evolution of concurrent agents

If two agents composed in parallel offer complementary communications, both can evolve to their next state:

$E_1 \equiv in'.m.E_1$
$E_2 \equiv m'.out.E_2$
$Src \equiv in.0$
$Sink \equiv out'.0$
$System_1 \equiv E_1|E_2$
$System_2 \equiv System_1|Src$
$System_3 \equiv System_1|Src|Sink$

Describe the evolution of $System_1$ in isolation; describe the evolution of $System_2$and $System_3$.

Linda mechanisms

Linda not a language but rather a conceptual approach to concurrency that relies on the notion of a *tuple space:* a shared collection of labelled data records. Example tuples:

```
("chan1", 1, 2, 3)
("chan1", 'a', 'b', 'c')
("cat", "grey", "5-toed")
("cat", "orange", "3-legged")
```

One way to think of tuple space is as an unordered communication channel.

Operations on tuple space: OUT, IN, RD, INP, RDP, EVAL. OUT places a new tuple in the tuple space:

```
OUT (tag, val1, val2, ..., valn)
```

IN removes a tuple from tuple space using *matching* and *assignment*

```
IN (tag, var-or-val1, ..., var-or-valn)
```

A tuple *matches* an IN operation if the tags are the same *and* the types of the fields of the tuple are the same as the types of the fields in the argument to IN *and* the tuple and the argument to IN have the same values in the positions that the argument to IN has values. Some tuple that matches the IN is removed from the tuple store and the values in the tuple corresponding to the variables in the argument to IN are assigned to those variables. If no such tuple exists, the IN operation blocks.

RD is similar to IN but does not remove the matching tuple.

RDP and INP are similar to RD and IN but do not block if there is no matching tuple.

Finally, EVAL is like OUT, but initiates concurrent evaluation of the tuple's fields:

```
EVAL (tag, f1(args1), f2(args2), ...fn(argsn))
```

A concurrent process will be started to evaluate the calls to f1, f2, ... fn. When it is finished the resulting tuple is placed in the tuple store.

Barrier Example

Suppose we need a barrier requiring 4 processes to arrive before any proceed further; We can initialize it with

```
OUT("barrier", 0)
```

Each time a process arrives at the barrier it executes

```
IN("barrier", ?counter);
OUT("barrier", counter+1);
RD("barrier", 4);
```

Question: why is there not a race condition in reading and updating counter in the first two lines?

Question: how could the barrier be reused?

Motivation for Linda

- *communication orthogonality:* neither senders nor receivers identify the other party to a communication (compare with channels)
- *space uncoupling:* when implemented in a distributed setting tuples deposited by any process at any location may be received by any process at any other location
- *time uncoupling:* a tuple added to the tuple-space will remain there until it is removed by IN. In some implementations this can extend to persistent tuple stores that last beyond the execution of all programs that use them
- *distributed sharing:* as seen above in the barrier example, Linda tuples can be used to simulate shared variables even in a settting where there is no shared address space.

Next time

MPI (end of chapter 7), Ada rendezvous and RPC

Assigment

Read Chapter 8