1 Chaotic elasticity

The elastic buffer presented in the course has a bad property from an implementation point of view: it can happen that the number of empty cells in the buffer grow even if no new element is added to the buffer. Could you see why ?

2 Controlling elasticity

We present now a solution to the problem that you have to implement in Scala using Pilib.

A buffer is modeled as a sequence of independent processes which can be of the four following types:

- Put: a process at the beginning of the sequence that can accept new items and create buffer cells for them.
- Cell(x): a cell of the buffer containing an item.
- End: a process that closes the sequence.
- Get(x): a process that can output x.

The expected behavior of the buffer is fully described by the following three *rewriting rules*:

$$\begin{array}{lcl} \mathrm{Put} & \xrightarrow{in(x)} & \mathrm{Put} \frown \mathrm{Cell}(x) \\ \mathrm{Cell}(x) \frown \mathrm{End} & \xrightarrow{\tau} & \mathrm{Get}(x) \\ \mathrm{Get}(x) & \xrightarrow{\overline{out}\langle x \rangle} & \mathrm{End} \end{array}$$

If a sub-sequence of the buffer matches the left-hand side of a rule it can be rewritten in the corresponding instance of the left-hand side.

The following rewriting sequence represents a possible execution of the buffer. For clarity the matching sub-sequences have been underlined.

$$\begin{array}{lcl} \underline{\mathrm{Put}} \frown \mathrm{End} & \xrightarrow{in(1)} & \underline{\mathrm{Put}} \frown \mathrm{Cell}(1) \frown \mathrm{End} \\ & \xrightarrow{in(2)} & \mathrm{Put} \frown \mathrm{Cell}(2) \frown \underline{\mathrm{Cell}(1) \frown \mathrm{End}} \\ & \xrightarrow{\tau} & \mathrm{Put} \frown \mathrm{Cell}(2) \frown \underline{\mathrm{Get}(1)} \\ & \xrightarrow{\overline{out}\langle 1 \rangle} & \mathrm{Put} \frown \underline{\mathrm{Cell}(2) \frown \mathrm{End}} \\ & \xrightarrow{\tau} & \mathrm{Put} \frown \underline{\mathrm{Get}(2)} \\ & \xrightarrow{\overline{out}\langle 2 \rangle} & \mathrm{Put} \frown \mathrm{End} \end{array}$$

3 Written exercise

How could you convince someone that your implementation actually models an unbounded buffer ? Is it equivalent to the implementation presented in the course ? Try to prove it. Write answers for these two questions.

4 Hints for the implementation

Recursive channels To implement the removal of a cell we need *recursive channels*, i.e. channels that can carry channels of the same type.

For instance, the declaration for the type of channels that can carry a pair consisting of an integer and a channel of the same type is:

class Channel **extends** Chan[Pair[int, Channel]];

A typical output on such a channel a will look like this:

a(Pair(2, a)) $* \ldots$

And a typical input will be :

a $*$ { **case** Pair(x, y) => ... }

Observing channels Now for observing a channel a you can attach to it a function which will be applied to the transmitted value each time a communication takes place along this channel. No other communication can take place before the observation function has completed. Example:

```
a.attach(x => System.out.println("a:" + x));
```