

Concurrent and Distributed Programming

Practical 2: Cellular Automata

Deadline: Week 8 practical session

Bernard Sufrin

12th February, 2008 (1.419)

Life

The Wikipedia entry for the best-known cellular automaton, which known as Life¹, and was invented by John Conway, describes it as follows:

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by loneliness.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours comes to life.

The initial pattern constitutes the “seed” of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed – births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

You may find it helpful or interesting to read the Wikipedia entry in full.

Code base for the Practical

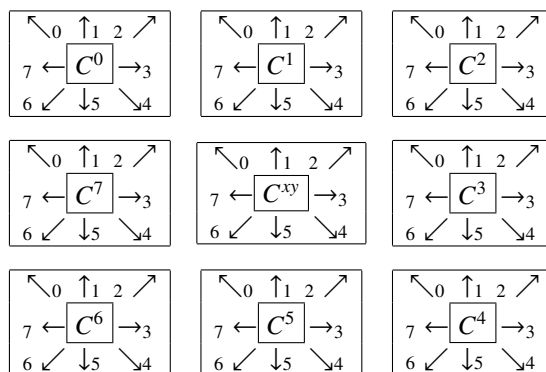
You are going to work with a concurrent simulation and visualization of an automaton that differs from Conway’s Life only in that the grid on which it evolves is finite. The

top and bottom edges of the grid are treated as neighbours, as are its left and right edges – in effect making it topologically a *toroid*.²

In the next section we provide an overview of the structure of the code. We recommend that you make yourself completely familiar with the source code, as well as the `cswing` library documentation before starting this practical. Both are provided on the course website.

Overview of the class Cellular

The diagram below shows the cell labelled C^{xy} connected through output ports numbered $C_0^x, C_1^x, \dots, C_7^x$ to its neighbours, labelled C^0, C^1, \dots, C^7 . The corresponding input ports are $C_4^0, C_5^1, C_6^2, C_7^3, C_0^4, C_1^5, C_2^6, C_3^7$.



A $cols \times rows$ cellular automaton evidently requires $cols \times rows \times 8$ inter-cell channels. The class `Cellular` sets up these channels, and constructs the cells of the automaton. In addition to `rows` and `cols` it is parameterised by output ports to the console and to the viewer, by an input port down which the viewer window reports events such as mouse clicks and by the physical size of the viewer window.

```
class Cellular( toConsole: ![String]
               , toViewer:   ![Instruction]
               , fromViewer: ?[Event]
               , cols:      int
               , rows:      int
               , dwidth:    int           // width of the viewer
               , dheight:   int          // height of the viewer
               )
{
```

The most complex aspect of the setup is the “wiring” of the cells to each other. First we generate the 3-D array of channels that will be used for this purpose:

```
val links : Seq[Seq[Seq[Chan[boolean]]]] =
  for (x<-0 until cols force) yield
  for (y<-0 until rows force) yield
  for (i<-0 until neighbours force) yield
    OneOne[boolean]( "Chan(" +x+"_" +y+"_" +i+"") )
```

The cell at (x, y) outputs to the output ports at $links(x)(y)$

```
val linkOut : Seq[Seq[Seq[![boolean]]]] = links
```

It will input from the output ports of its neighbours. Calculation of the coordinates of its neighbours is moderately straightforward – it is just necessary to use arithmetic modulo the size of the cell array:

```
val linkIn : Seq[Seq[Seq[?[boolean]]]] =  
for (x<-0 until cols) yield  
for (y<-0 until rows) yield  
{ val left = (x+cols-1) % cols  
  val right = (x+1) % cols  
  val up = (y+1) % rows  
  val down = (y+rows-1) % rows  
  val nx = Array(right, x, left, left, left, x, right, right)  
  val ny = Array(down, down, down, y, up, up, up, y)  
  for (i<-0 until neighbours) yield links(nx(i))(ny(i))(i)  
}
```

Next we set up the collection of cells

```
val cells =  
  for (x<-0 until cols force) yield  
  for (y<-0 until rows force) yield  
    new Cell(toViewer, cellVisual _, x, y, linkIn(x)(y), linkOut(x)(y))
```

The automaton is brought too life by running the concurrent composition of its cells.

```
val runCells = || (for (row<-cells; cell<-row) yield cell.process)
```

But it must also handle messages from the viewer that report mouse presses made on images of individual cells.

```
val listenToViewer =  
proc  
{ repeat  
  { val event = fromViewer ? ;  
    event match  
    { case MouseHit(_, _, _, mods, hits) =>  
      for (thing<-hits) mouseHit(thing.asInstanceOf[Cell])  
      case other => { }  
    }  
  }  
}
```

It therefore exports

```
val process = runCells || listenToView
```

Our prototype also has a method that seeds it with a glider shape:

```
def seed =  
{  
  for ((x, y) <- List((0,0), (1,0), (2,0), (0,1), (1, 2)))  
    cells(x)(y).setState  
}
```

This should be called before the automaton is started, for it is not thread-safe.

Questions:

1. Why do you think a cell reads its neighbours' states in parallel with writing its own state to its neighbours?
2. Why do you think a cell writes its own state sequentially to its neighbours? Could it do the writes in parallel?
3. Why do you think `seed` should be called *before* the automaton is started?

Overview of the Cell class

In the starting code base for the practical, each cell is modelled by a `Cell` object. Cells share a channel to the viewer and a method of generating visual images of themselves. A cell also knows its position in the grid, and has connections to and from each of its eight neighbours.

```
class Cell( toViewer:    ![Viewer.Instruction]
            , cellVisual: ((int, int, Color)=>Visual)
            , x:int, y:int
            , fromNbr:    Seq[?[boolean]]
            , toNbr:      Seq![boolean]
            )
extends Viewer.Thing
{
```

The `Cell` class inherits from `Viewer.Thing` because the cells co-operate with a `Viewer` component to visualize themselves.³ Every `Thing` has an `image : Visual` method that – in effect – generates a description of its current appearance and location. For cells this method is defined by:

```
var state                = false
val deadImage : Visual = cellVisual(x, y, GRAY)
val liveImage  : Visual = cellVisual(x, y, GREEN)
def image      : Visual = if (state) liveImage else deadImage
```

The behaviour of the cell at (x, y) in the grid is simulated by an exported cyclic process:⁴

```
var animating = true
val process =
  proc (format("Cell(%d,%d)", x, y))
  { toViewer!Alter(this)
    repeat (animating) {
```

At the beginning of each cycle, it sends its state to each of its neighbours, and records their current states:

```
exchangeState()
```

Then it calculates its next state, and if necessary informs the viewer that it has changed state.

```
val nextState = calculateNextState
if (nextState != state)
{ state = nextState
  toViewer!Alter(this)
  toViewer!Display
}
}
```

The state exchange method reads concurrently from all its neighbours while writing sequentially to them:

```

val neighbours      = 8
val neighbourState = new Array[boolean](neighbours)
val exchangeState =
  (proc { for (nbr<-toNbr) nbr!state }
    || || (for (n<-0 until neighbours) yield
          proc { nbrState(n) = fromNbr(n)? })
  )

```

Calculation of the next state begins with a count of the number of live neighbours:

```

def calculateNextState =
{ var count = 0
  for (s <- neighbourState) if (s) count += 1
  ... calculate the next state ...
}

```

Overview of the main program

The main program is defined by the module (object) `lyfe`. It begins by setting up the `ox.cswing.Viewer` component that will be used to visualize the automaton, and the channels that will be used to send instructions to it and to receive events from it.

```

object lyfe
{ def main(args: Array[String])
  { val toViewer = ManyOne[Instruction]
    val fromViewer = OneOne[Event]

    val viewer = new Viewer("View", toViewer, fromViewer)
      { reportHits = true
        antiAlias = false
      }
  }
}

```

Apart from reading arguments from the command line and setting up the user interface, all that remains for the main program to do is to construct the cellular automaton, and fire up the relevant processes.

```

var cols, rows = 11
var width, height = 200
for (arg<-args) ... read cols, rows, width, height ...

val cellular =
  new Cellular(toConsole, toViewer, fromViewer, cols, rows, width, height)
  cellular.seed

val app = new JApplication("CellularAutomaton") { ... set up GUI ... }

( viewer.process
  || console(toConsole)
  || cellular.process
  )()
}

```

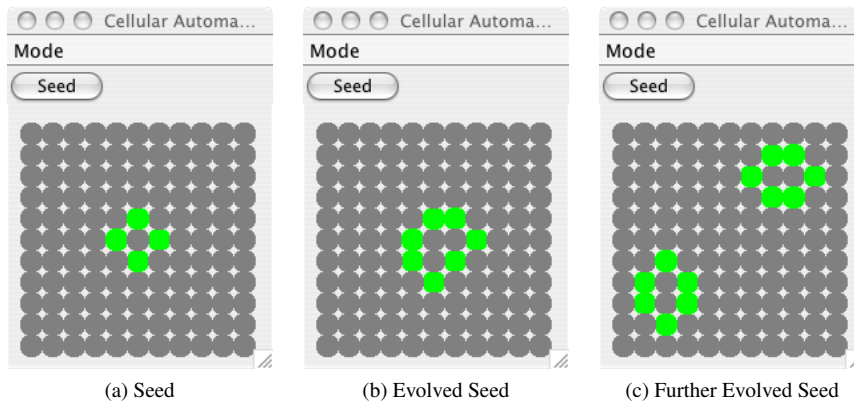


Figure 1: Snapshots of a cellular automaton

Tasks to be accomplished in the Practical

TASK 1: Compile and run the initial code base

This task speaks for itself. If you find it hard to do this then you may find it impossible to accomplish any of the subsequent tasks.

If you have downloaded the CSO, and CSWING libraries you should be able to run the program with a command line of the form:

```
scala -cp cso.jar:cswing.jar :. lyfe
```

This should generate the image of an automaton with a static “seed” on it, something like the configuration shown in figure 1a.

Now press on the dead cell in the middle of the seed. This may cause the automaton to evolve to the static configuration shown in figure 1b, whence a press at its apex will cause it to evolve to the static configuration shown in figure 1c.

But this is not the case on all computers and with all grids. The fact is that each cell is evolving at its own pace, and that the evolution of the automaton as a whole is nondeterministic. The implementation you have been given has *bugs*!

TASK 2: Synchronize the Automaton

On most machines you will observe that the effects of clicking on a cell, or of clicking on the seed button once the automaton has started are very unpredictable.

On some machines you may notice that the cell images are not being updated simultaneously. This is because *each separate cell* sends `Display` messages to the viewer when it has changed its appearance, when all that is needed is for a single such message to be sent at the end of each generation.

Your first substantive task is to arrange that cell images are updated *synchronously*. There are several ways in which this might be done.

The simplest way to begin is to interpose a process between the cells and the viewer that counts `Alter` messages before forwarding them to the viewer. Every $row \times col$ messages it should interpose a `Display`. Cells would have to be modified to send `Skip` messages when they don't change.⁵

Unfortunately this isn't quite good enough, for once a cell has sent its `Alter` or `Skip` message it is free to continue again, and it might well evolve to another state before all other cells in a generation have finished their cycle. The remedy for this is to add a one-one synchronisation channel to each cell that is read by the cell at the bottom of its cycle, and for the counter to arrange for synchronisation signals to be sent down these channels at the end of each generation. [Hint: it is probably essential it to have the sync signals sent by a separate process, rather than sending them itself.]

TASK 3: Single-step the Automaton

Now that the automaton has been synchronized, it should be possible to add a mode of operation in which the user has to press a button in order for the automaton to evolve through a single generation. Add such a mode and such a button. It should be possible to change modes at any time (between generations). One possible way of structuring this would be for the process that distributes synchronisation signals to behave differently depending on what mode the automaton is in: in order to avoid race conditions it might make sense for this process to receive messages from buttons in the GUI as well as from the

TASK 4: Detect stabilization of the Automaton

If the automaton goes through a single generation without the appearance of any cell changing, then it is in a stable state. When it is in such a state it should be stopped – or (to be precise) the individual cells should be stopped. Arrange for this to happen. One possible way of doing this would be to arrange for the synchronization channels to be used to send “halt” as well as “continue” signals.

TASK 5 (Optional): Clear, Edit and Restart the Automaton

Careful design of the machinery you used in tasks 2, 3, and 4 should enable you to edit the automaton (by clicking on cells) when the automaton has stopped or when it is waiting for the single-step button to be pressed. You should also be able to add a clear button that kills all live cells.

Optional: Variations on Life

Experiment with the detailed behaviour and appearance of the cells if you have time. According to the Wikipedia article:

Since Life's original inception, new rules have been developed. The standard Game of Life, in which a cell is born if it has exactly 3 neighbours, stays alive if it has 2 or 3 living neighbours, and dies otherwise, is symbolised as "23/3". The first number, or list of numbers, is what is required for a cell to continue. The second set is the requirement for birth. Hence "16/6" means "a cell is born if there are 6 neighbours, and lives on if there are either 1 or 6 neighbours". HighLife is 23/36, because having 6 neighbours, in addition to the original game's 23/3 rule, causes a birth. HighLife is best known for its replicators. Additional variations on Life exist, although the vast majority of these universes are either too chaotic or desolate.

...

Immigration is a variation that is the same as Life, except that there are two ON states (often expressed as two different colours). Whenever a new cell is born, it takes on the ON state that is the majority in the three cells that gave it birth. This feature can be used to examine interactions between spaceships and other "objects" within the game. Another similar variation, called QuadLife, involves four different ON states. When a new cell is born from three different ON neighbours, it takes on the fourth value, and otherwise like Immigration it takes the majority value. Except for the variation among ON cells, both of these variations act identically to Life.

Notes

Note 1 http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Note 2 In this respect, as in many others, our implementation derives from that presented in *Geraint Jones and Michael Goldsmith, Programming in occam 2*. This book is now out of print, but is available in pdf form on the course Website. The relevant chapter is presented there as a separate file, for interest.

Note 3 As will be clear from the documentation of `Viewer` and its associated classes, a message of the form `toViewer!Alter(thing)` tells the viewer that is listening to the port at the other end of the `toViewer` channel that the `Thing thing` has changed its state, and should be redrawn at the next `Display` event. A message of the form `toViewer!Display` tells the display to redraw the changed parts of its display.

Note 4 The process corresponding to cell (x, y) shows up in deadlock and stack back-traces as a thread named `Cell(x, y)`.

Note 5 It is worth noticing that when processes start up they send an `Add` and a `Display` message to the viewer. These should be passed on, but not counted, by the counting process.