Advanced Functional Programming Project - Pasture in Erlang

Carl Carenvall & Emil Wall

December 29, 2012

Abstract

Ecosystems are a big research area in biology and related fields, where interactions between species and their effect on the overall system is observed. This report describes a simple simulation of an ecosystem, modelled as fixed objects and live creatures in a two-dimensional grid. The simulation is tick-based and each object is implemented as an Erlang process.

Contents

1 Introduction

The pasture is built around a two-dimensional delimited area of a rectangular form. We view the pasture from above, and ignore any hills and slopes. The rectangular area is divided into a grid. Each creature (animal or plant) and object included in the model represents a *population* of creatures or objects of the same kind. This allows the creatures to reproduce without having another creature of the same kind nearby. Each object and creature in the pasture occupies exactly one square.

This dynamic model of the ecosystem of a pasture is of course very simplified compared to reality.

2 Related Work

Ecosystem simulation, game of life, multiprocessing, message passing

3 Requirements

3.1 Functional Requirements

1. The pasture should contain fences, grass tufts, foxes and rabbits.
2. The simulation time is measure in the unit *glan* (an ancient unit of time which is at least 200 000 glan old and nowadays almost completely forgotten).
3. Animals, that is, foxes and rabbits, eat food at adjacent squares.
4. It takes a certain number of glan for the animal to move one square.
5. If a square is occupied by a fixed objects nothing else is allowed to ever enter the same square.
6. Fences are the only type of fixed objects.
7. Live creatures can reproduce. They die if they are without food too long or get eaten by another creature.
8. Plants are live creatures that reproduce whenever there is room for them to grow.
   (a) Grass tufts are plants that reproduce by division. They can grow whenever there are no fixed objects.
   (b) Grass tufts divide in two after a certain time (given as parameter), provided that there is a free adjacent square.
   (c) Grass tufts do not eat.
   (d) Grass tufts can not starve to death.
9. Animals are live creatures that can reproduce at a rate depending on the type of animal, under certain circumstances.
   (a) An animal can live a certain number of glan before it dies of starvation.
   (b) An animal can move around in the pasture, at different speeds for different types of animals.
   (c) Animals may only move to an adjacent square, either vertically, horizontally or diagonally.
   (d) An animal eats food if it is hungry and the food is at an adjacent position.

(e) An animal reproduces when it eats, but it can only reproduce when it has reached a certain age, and after a reproduction it takes a certain time before it can reproduce again.

(f) Animals can only reproduce if there is a free adjacent square

(g) Animals live until they die of starvation or get eaten.

(h) Rabbits are animals that have foxes as enemies and eat grass tufts.

(i) Foxes are animals that do not have any enemies and eat rabbits.

3.2 Non-functional Requirements

1. Each animal, plant and fence should be implemented as an Erlang process.
2. Each position in the pasture may only contain one object or creature.
3. The size of the pasture, the initial number of inhabitants of different types and the behavior of the different animals at the start of the simulation is controlled with parameters that can be set at the start of the simulation, without re-compiling the program.
4. Surrounding the pasture with fences is sufficient to keep the inhabitants inside, there is no need for (and must be no) knowledge of the size of the pasture.
5. The inhabitants of the simulation must have a position so that the system knows where they are, but they must not use this position in their intelligence (if they are intelligent).
6. All parameters should have default values.
7. The default parameters should be set so that the system is reasonably stable, so that there isn't one species which always dies out or floods the pasture.
8. Magic numbers and hard coded constants should be avoided to the extent possible. If used, they should be properly defined as descriptively named constants.
9. The program should be free from duplicated code, for instance in the different animal classes (but the requirement applies to the simulation code in general).
10. The program should not violate the plagiarism principles of the Department of Information Technology.

4 User guide

4.1 Getting Started

TODO: Dependencies, how to run a simple simulation, how to set parameters, how to run tests

4.2 Design

TODO: How we arrived at our solution, how we use any given code, how we use standard libraries. Description and motivation of design decisions.

TODO: Algorithms and representation of data.

4.2.1 Communication protocol and grid representation

The entities (fixed objects and live creatures) need a way of communicating with each other, preferably in constant time with its neighbours. Therefore, the grid is represented by an ets that maps grid coordinates (x, y) to entity PIDs.

Example:

A fox (randomly) selects an adjacent position in the grid to move to, and performs a lookup in the dict (which maps positions to entity PIDs) to get the PID of the process located there, if any. If there is no process there, the fox moves there and updates its state and position in the dict. Otherwise, it sends a message to the process requiring it to identify itself. If the process answers and says that it's a sheep, the fox devours it before moving there. If it says that it's another fox or a fence it stays at the current position and only updates hunger and similiar stats. If it says that it's a grass tuft it moves there as if there was nothing there. The dict will have to store a list of PIDs to accomodate the need of having both a fox and a grass tuft on the same position in the grid.

4.2.2 Objects and classes

4.3 Known Limitations

We were able to simulate instances of size ... (TODO) with the other parameters set to their default values. The different parameters influences complexity and

speed of convergence in the following way: TODO

It would be possible to add intelligence and vision to the simulation, so that animals can see food or threats at a distance and act accordingly. This has not been implemented. Animals would then tend to move toward food and run away from enemies. Different types of animals would have different sight ranges, determining at which distance they can discover food, obstacles and/or enemies.

The architecture supports adding more types of fixed objects, animals and plants but we decided against doing this due to time constraints.

TODO: Details on failing tests and lacking functionality, and reasons for why

4.4 Detailed Documentation

4.5 Complexity analysis

5 Discussion

TODO Difficult corner cases, anything that required deliberation on our part, or any unexpected solutions to sub-problems.

TODO Race condition considerations

TODO An argument to why our solution produces a correct result.

FIXME Foxes can eat very fast, a more realistic scenario would require them to get hungry again before they eat again

5.1 Grid representation

The entities (fixed objects and live creatures) need a way of communicating with each other. One way to do this would be to use a key-value data store structure such as a dictionary (dict). The dict could then map grid coordinates (x, y) to lists/sets of entity PIDs.

An alternative design would include not having a dict process but instead representing the grid using processes with links to its neighbors, each representing a tile/position. This would mean initializing the grid as a two-dimensional, doubly-linked list, but might improve scalability. One could also consider having multiple

dict processes, to reduce the bottleneck factor, each responsible for different parts of the grid.

Example (using dictionary):

Example (using processes):

The initialization is complicated compared to using a dictionary. For each entity that is spawned, it needs to know the PID of the process representing the position, and processes for adjacent tiles must be spawned when needed.

6 References

7 Appendix

TODO: code listings