Christopher Pauliks and Utsav Pandey
Comp 473
Project 3 Report
December 14, 2011

While individual objects would be sufficient for a simulation such as the one developed for project three, using actors presented obvious advantages. Actors further encapsulate the individual animals, increasing their independence. As an added benefit, they can act concurrently, increasing performance. Our implementation consists of three major object types: the animals, Hare and Lynx, which act similar to the animals in the NetLogo version, and a World, which manages the Hares and Lynx. Specifically, it sends commands to all the animals and uses their responses to update its main data structures, all of the actors in the simulation and their locations. All messages to animals are case objects. All messages from animals are case classes only containing the location of the actor as Ints or a reference to an actor, if necessary. These restrictions prevent mutable state from being sent to other actors.

Due to the use of actors, some changes had to be made to ensure that the simulation functions correctly, as we cannot guarantee the order in which messages will be received. It would be foolish to send another command to an animal before we had confirmed it had received and acted on the previous one. To solve this, we split each World clock tick into four stages, `Move`, `Eat`, `Reproduce`, and `Age`, corresponding roughly to the five stages of the NetLogo version. Each stage has its own confirmation and the World expects a reply from each animal before moving on to the next stage. Originally, this created a possible livelock condition when an animal died of natural causes but the World did not receive this message until after all other animals had confirmed their work completed. As a result, the World would never move to the next stage as it was waiting for the last confirmation from a dead animal. The solution was to check whether the stage is complete after any animal sends a death message, just in case it was the last one it needed.

We chose to implement project three using Akka's Actor library instead of Scala default Actor classes. This choice provided both benefits and drawbacks, as some parts of the implementation were simplified with Akka, while others were made more difficult. The actors themselves were simpler to develop. Instead of using `receive` or `react`, Akka actors only have a `receive` function, which can function like the react loop while looking more closely to the receive function, making the code more readable. Akka also implicitly includes a reference to the sender of a message with every message. This makes callbacks easier, as an actor can respond to a message by calling the function `reply` from within the `receive` cases, instead of having a reference to the sender included in the original message as an address for the response. Finally, Akka enforces total actor independence. The only way to create a new actor is from the `actorOf` factory methods. These return a threadsafe ActorRef, or actor reference, to the new actor, instead of the actor itself. With an ActorRef, you can only start of stop the actor's message queue. Attempts to create new Actors outside of the factory methods will throw exceptions. Due to these constraints, the only way to change the state of an Akka actor is by sending it messages.

These advantages come with one major disadvantage. Akka actors are harder to unit test because of their obfuscation via ActorRefs. There is no way for a normal test case to see the internals of the Actor and test methods and functions inside. However, Akka does provide an actor-tests library to facility testing. To test the replies from an actor, the TestKit trait transforms a unit test class into a mock actor. The test class can then send a message to an actor and, using the `expectMsg` functions, ensure that a correct response is received. And in order to test the methods within an actor, the library includes a `TestActorRef` factory method that produces an ActorRef which can have its underlying actor extracted via its `underlyingActor` value.

Even with the increased readability of Akka actors, it is still somewhat difficult to read and reason about actor classes. Akka's actor independence does give us two good places to begin looking. After an actor is constructed, the first code that can be run is within the Actor's `preStart` method, an overrideable method from the actor class and is guaranteed to be executed after the actor is started but before the message queue has started. While the message queue is running, the only code paths executed by the actor have to start with the actor's `receive` method. By looking at the message cases, you can determine what the actor is doing, following any methods or functions called within `receive`. If the actor sends a reply or another message somewhere on this path, you can do the same analysis for the other actor's class. Finally, when an actor is stopped, the last code run is in the Actor's overrideable `postStop` method.

Actors are a much simpler concurrency solution than threads requiring locking and synchronization. Still, any concurrency solution is more complex than a non-concurrent program. While deadlock and livelock are less likely to occur with proper actors, care must still be taken to ensure each actor is doing the right work at the right time. In many situations, such as with our simulation, the benefits of actors outweigh the risks and increased complexity of a concurrent implementation.