0.1 Requrements

0.1.1 Aesthetic Requirements

The requirements therfore on this project are to produce an ant colony simulation which exploits the parallel benifits of the purely functional programming language Haskell. The behaviour of the project's resulting artifact aims to portray a functioning ant colony. Ants should seek out food using exploratory strategies, communicate locations of food to other ants in the colony using pheremones and return to the colony's nest with food. The ant colonies increasing size should be reflected in the size of the nest, and the nests size in turn will dictate the amount of new ants generated by the colony. The nest's physical size will grow relative to the amount of food successfully brought back to the colony by the ants in the simulation.

0.1.2 Exploratory Strategies

Ants will explore the simulation world by moving randomly away from the nest, each ant will have a direction property which when set allows them to track their location relative to where they have already been. One example of the use of storing directions within the ant is to detect when an ant has made a circle and returned to a location it has been before, the An ant will then be able to make the decision not to take the same path as it did previously leading to slightly more intelligent exploration strategies. Another example of an exploration strategy is moving away from ants which are not holding food and going in the oppsite direction to ants who are carrying food, if ants carrying food are looking to return to the colony this would also be an effective strategy.

0.1.3 Communicating locations of food

Once an ant locates food it will excrete pheremone which will be left on the Ant and on the current location where the food has been located. Ants will have the ablity to ?absorb? pheremone from their current location, this coupled with the surface of the simulation disapating pheremone every simulation step will allow ants to leave a dynamic path towards food sources by leaving a trail of pheremone. Once ants who are currently searching for food detect a pheremone indicating food they would be able to adjust their exploration strategy to detect which surrounding cell has the greatest pheremone level and use this trail to find food more efficiently.

0.1.4 Returning to the colony

This requirement could also be achieved through the use of a different pheremone, or by allowing ants to retrace their steps. The ideal behaviour of the simulation would use a mixture of the two as ants are unlikely to remember the entire path back to their nest.

0.1.5 Nest size

The nest size will increase relative to how much food is brought back to the nest. The ants task of bringing food back to the nest enables the colony to fulfill its task of increasing in size.

0.1.6 GUI

Set parameters View the simulation

0.1.7 Technical Requirements

Above all the aim of the project it to exploit the parallelism exposed by pure function in Haskell. The project could be also expanded to make use of more than one processor by way of distributed computing. The resulting artifact will be tested in three ways in order to determine whether it has made effective use of parallelism within the simulation. Firstly attempts will be made to produce functions which run the simulation in a serial manner in order to compare the runtimes of the parallel design with the serial. Secondly, varying loads will be placed on the simulation, this will be achieved by creating worlds dynamically from a set of parameters. Thirdly, the simulations design must allow for varying degrees of parallel granularity. The parameters introduced from these different levels of granularity can then be used to ?fine-tune? the simulation to determine optimum settings and better assess where parallelism makes improvements on run time and where it does not.

0.1.8 Algebraic Test Driven Development

It will be critical to test all code as it is produced, this not only saves time debugging but testing the final project build is more likely to be successful. This approach to software development is applied by holding to a short development cycle in which the developer writes a test case for a new instance of functionality within the software, then produces code that will pass that test before refactoring the resulting code as necessary. [Test Driven Development by Example Beck] To some respect the language Haskell, in which this project will be undertaken, allows this approach to development to be taken naturally through it's rich and powerful typesystem. The use of type signatures as annotations to functions allows the programmer to specify the type of values the function should excpect as parameters and the expected type of the return value.

The Quick Check package

Test completeness