CS429 Final Project: Frogs and Toads

Dachuan Yu
dachuan.yu@yale.edu

April 29, 2001

My final project of Functional Programming consists of the following parts:

1. Write a haskell program to implement a general search algorithm – depth first search;
2. Use this general search program to solve a small puzzle – **Toads & Frogs**;
3. Use SOEGraphics to demonstrate the solution of the puzzle.

1 Search Program

In the first part of the project, I implemented a general search algorithm, namely depth first search.

First of all, I implemented the (directed) graph abstract data type `Graph n`, where `n` is the type of the nodes. It is defined as follows using the edge list representation.

```
type Graph n = [(n, [n])]
```

I assume `n` be an instance of the `Eq` class, because I want to compare the equality of nodes. Given this type, the operations needed to manipulate the graph are:

- `mkGraph :: Eq n => [(n, n)] -> Graph n`
  takes the list of edges (each edge consists of the two end vertices) and returns a graph; note that the graph is directed.
- `adjacent :: Eq n => Graph n -> n -> [n]`
  returns all the neighbours of a node in the graph.
- `nodes :: Eq n => Graph n -> [n]`
  returns all the nodes in the graph.
- `edges :: Eq n => Graph n -> [(n, n)]`
  returns all the edges in a graph.

Since searching takes place over a large graph, it is not desirable to store this entire search space in memory. For efficiency reasons, it is better to examine one node at a time, then generate adjacent nodes from the value of that node. Therefore, I decided to work on an implicit graph. All we need is the following information:

- an appropriate type `node` which represents node information;
- a successor function of type `node -> [node]` which generates the list of successors of a particular node;
- a goal test function used to check the current status;
- and a start node to start the search with.

Based on the above, our depth first search function is as follows:

```
depthFirstSearch :: Eq n => (n -> [n]) -> (n -> Bool) -> n -> [n]
depthFirstSearch succNodes goalTest startNode = ...
```

2 Puzzle

The *Toads And Frogs Puzzle* is also known under the names of *Hares and Tortoise* and *Sheep and Goats*. With no animals at hand, it can be played with two kinds of coins. The following names still reflect on the essence of the activity: *Hop, Skip, Jump* and *Traffic Jam*.

N frogs are placed on N successive positions on the left of a string of squares; M toads occupy M rightmost squares. On the whole, there are $M + N + 1$ squares, so that just one square remains unoccupied.

Frogs only move rightward; toads move leftward. Every move is either a **Slide** to the nearby square or a **Jump** over one position, which is allowed only if the latter is occupied by a fellow of a different kind. In any case, no two animals are allowed in the same square.

The goal is to move toads into M leftmost positions and the frogs into N rightmost positions.

This puzzle is pretty simple. However, it is enough to demonstrate the idea of representing a puzzle as a search problem and using a general search algorithm to solve it.

The representation of the puzzle as a search problem is essential to solving the problem. Basicly we want to define the following concepts:

1. State space (nodes).
   Each node (configuration) is represented as a list of integers.
   `type Node = [Int]`
   The list stands for the board. Integer 1 stands for Frogs, integer -1 stands for Toads, and 0 stands for the unoccupied square.

2. Moves (edges).
   A single move either slides a creature adjacent to the unoccupied square, or jumps a creature over another onto the unoccupied square. It is implemented as the `succs` function.
   `succs ::  Node -> [Node]`

3. Start state (initial node).
   The initial configuration is represented accordingly.

4. Goal test (goal node).
   There is only one goal node in this game. It is represented accordingly. Since we require our node data type to be an instance of the `Eq` class, `(==goal)` serves as the goal test, where `goal` is the goal node.

3 SOEGraphics Demonstration

Our depth first search function keeps track of the path leaded to the solution. Once a solution is found, we use SOEGraphics to demonstrate the answer. It is implemented as the `gameShow` function.

Inside `rugs`, after loading the program `frogs.hs`, one can type in `gameShow` to activate the game show window. The game show window is a rectangle window named “Frogs and Toads”. There are a list of squares in the window. The number of squares is determined by the game configuration. The green squares on the left stand for frogs, and the blue ones on the right stand for toads. The white square in the middle stands for the empty square. There is also a red indicator below the squares. It shows the users the next move. One end of the indicator always points to the white square no matter where the white square is in the current step. The other end of the indicator indicates the object which is going to move onto the current white square in the next step. The solution moves are demonstrated by pressing down any key. Once a key is pressed, the game moves to the next step according to the solution. After the solution is fully demonstrated, one more key press would close the window.