N-dimensional Tic Tac Toe, and Adventure in Modules

Alex Grasley Jeff Young Michael McGirr

1 Introduction

2 Overview of Project

Our project initially began as a pokemon simulator - but early on we realized that we could make better use of the SML module system by approaching the problem of simulating player-based games in a more abstract general way. By doing so we could define the basic notion of what a game simulation requires and isolate a pattern to follow for any number of games that fit this model. A game would then be a specific implementation - in our case *Tic-tac-toe* - that used this pattern.

3 Program description

Our project is separated between the code that describes the game simulation and the code that uses this to make a specific *Tic-tac-toe* implementation. The game simulation is located in the `game.sml` file. Likewise, the code for the *Tic-tac-toe* implementation is located in `tictactoe.sml` and uses modules from `matrix.sml`.

As we touched on before - the notion of a game is generalized in the `game.sml` file. This defines the signatures and functor to run a game. The general pattern for a game under our model consists of three purposely isolated pieces. These are *State*, *Actions*, and *Agents* which are each given their own signature in `game.sml`. These are then wrapped together with a functor. The idea is that any game consists of a state, a set of actions on that state, and an agents that take actions for a state. A specific implementation of a game would use the relationship between these three general pieces to define a runnable game with its own modules.

Our implementation of *Tic-tac-toe* uses this game model to operate and is primarily defined within `tictactoe.sml`. Here we have two signatures - one that extends state (the `STATE` signature from `game.sml`) for *Tic-tac-toe* called `TTTSTATE` and another that does the same with action called `TTTACTION` - we don't extend the `AGENT` signature specifically for *Tic-tac-toe*.

There are many module structures instantiated here for the various kinds of state, actions and agents that we eventually want to use - such as `TttState` and `TttAction` for instance. These are bound to functors like `TttStateFn` which takes a module implementing a square matrix module (`SQUAREMATRIX`) and returns a module that implements `TTTSTATE`.

The functor `TttActionFn` (used by modules structures like `TttAction` and `Ttt3DAction`) takes a module that implements `TTTSTATE` and gives us a module implementing `TTTACTION`. The functors `TttRandomAgent` and `TttHumanAgentFn` take modules which implement `TTTACTION` and return a module that implements `AGENT`.

We can then instantiate a structure like `TttExecRandom` that will run a game of *Tic-tac-toe* by providing our `ExecFn` functor from `game.sml` with a module that implements `AGENT`. `ExecFn` then gives us a module that implements `EXEC`.

The idea being that we instantiated specific module structures (like `TttState`) for different cases of actions and states that may occur in various kinds of *Tic-tac-toe* games. We then described functors which linked these structures to the functions meant for that specific use and returned a new . When provided with the correct input module these functors will give us modules that implement that portion of the game pattern - which we then use with another structure and another functor for the next modular piece of the game.

4 Design Decisions

4.1 Creating an Abstract Game Engine

4.2 Higher Ordered Signatures, and the "Include" incantation

4.3 Separation of IO, or How I learned to not fight SML in search of Purity

4.4 You can do it in 2-dimensions, but can you do it in n-dimensions!

4.5 The Functor is love, the Functor is life