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 which make up the pattern of a game and a 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 to execute a game. 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`. This roughly follows the order: State $\rightarrow$ Action $\rightarrow$ Agent $\rightarrow$ Execution.

In `tictactoe.sml` 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 (the `ACTION` signature from `game.sml`) called `TTTACTION`. We don't extend the `AGENT` signature specifically for *Tic-tac-toe*.

There are many module structures instantiated in `tictactoe.sml` for the various kinds of state, actions and agents that we eventually want to use - such as `TttState` and `TttAction` for instance - but each will be used to implement the next kind of module. These module structures are bound to functors like `TttStateFn` which takes a module implementing a square matrix module (`SQUAREMATRIX`) and returns a module that implements `TTTSTATE`. The functors in `tictactoe.sml` effectively take a module structure that implements the previous module of the game pattern and produce the current (or next) modular piece.

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` for example) that will run a game of *Tic-tac-toe* To do so we provide 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 states, actions and agents 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 module. When provided with the correct input module, these functors will give us modules that implement that portion of the overall game pattern.

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

In the first implementation of our project we simulated Haskell's IO monad by creating a "show" signature, and an "IO" functor, as shown below:

```
signature SHOW =
sig
    type a
    val show : a -> string
end

signature IO =
sig
    structure S : SHOW

    val printIO : S.a -> unit
    val read : 'a -> string option
    val say : string -> unit
end

functor Io (structure Sh : SHOW) : IO =
struct

  structure S = Sh

  (* append a new line to a str, this is expensive *)
  fun appendNewLine str = implode $ (explode str) @ [#"\n"]

  fun printIO x = print o appendNewLine o S.show $ x

  (* function to get user input, it doesn't do anything with its argument *)
  fun read _ = TextIO.inputLine TextIO.stdIn

  fun say str = print o appendNewLine $ str

end
```

We decided against this approach i.e. isolating IO into a separate module, because it began to pollute our design's dependency graph with unwanted edges. In haskell type class instances exist in an global overloaded name space, that takes advantage of haskell's dispatch system. So each data type that is implemented for the `Show` typeclass is able to be printed to StdOut *without* carrying around its respective show function. This is not the case in SML and it was sorely missed in our implementation. If we were to follow the Haskell style of IO (separating impure and pure code explicitly), then an IO dependency would be requisite anytime we needed to perform any IO. Thus, the IO node would become a dominating node in our dependency graph and the import of it into every functor would quickly become unweidly and boilerplate. Hence, we chose an SML style design where the structures that are mapped upon carry any relevant functions with them to maintain a clean decomposition in our design. This type of structure bloat, although disgraceful to a haskell programming, seems to be encouraged by SML's module system.

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