N-dimensional Tic Tac Toe, and Adventure in Modules

Alex Grasley Jeff Young Michael McGirr

1 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.

2 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 agents that select which action to perform on a given 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 → Action → Agent → 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*. These extended signatures are perfectly compatible with the original signatures due to the structural matching of SML signatures.

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 module structures like `TttAction` and `Ttt3DAction`) takes a module that implements `TTTSTATE` and gives us a module implementing `TTTACTION`. The functors `TttRandomAgentFn` 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.

3 Design Decisions

3.1 Creating an Abstract Game Engine

3.2 Exploiting Structural Typing

One major element of SML's module system that we were about to exploit in our project was the structural matching of modules to signatures. Coupled with liberal use of the `include` keyword, this allowed us to easily extend signature definitions in a way reminiscent of inheritance and subclassing in OOP.

This design decision is most apparent in our extension of the abstract game engine to our chosen implementation domain of *Tic-tac-toe*. Our more abstract interfaces defined in `game.sml` were often insufficient for describing the full interface needed for our concrete implementation. For example, *Tic-tac-toe* state modules needed to export a `cell` datatype that represented the actual Xs and Os placed on the game board. Including this datatype in the abstract `STATE` signature in `game.sml` would be inappropriate, as any game outside of *Tic-tac-toe* has no need for the `cell` datatype. Instead, we created a new signature `TTTSTATE` that extended the `STATE` signature via the `include` keyword with additional interface elements. Additionally, this extension refined some of the abstract types in `STATE` using the `where` keyword in order to give them concrete type definitions. For example, the abstract type `state` is defined to always be equivalent to the type `cell * (cell Matrix.container)` in a module implementing `TTTSTATE`.

We were able to exploit this refined state signature to narrow the domain of the `TttActionFn` functor to only states appropriate for *Tic-tac-toe*, rather than any abstract game state. However, thanks to the structural matching properties of the SML module system, we can still use a module implementing `TTTSTATE` wherever we would use a module implementing `STATE`. In fact, because the matching is structural and not nominal, we get this property for free, with no need to specify the relationship by name in the code. This allows us to still use signatures and functors in our implementation that depend only on the more abstract definitions like the `AGENT` signature or the `ExecFn` functor, which are only defined for the more abstract game engine.

3.3 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 its import into every functor would quickly become boilerplate and unweidly. Hence, we chose an SML style design, where the structures that are mapped upon carry any relevant functions with them, which maintains a clean decomposition in SML. This type of structure bloat, although disgraceful to a haskell programming, seems to be encouraged by SML's module system.

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

Our initial intent was to implement a pokemon battle simulator. As we dove into the problem domain we realized that instead of falling into a specific, flow-chart design with modules like: Pokemon, Combat, Moves etc. it would be better design to create an abstract game engine and then make pokemon *an instance of* the abstract game engine. A similar pattern occurred during our first implementation of the matrix library utilized by the tic tac toe state. We wrote a single matrix module with more rigidly defined dimensions and underlying implementation details. While it was abstract, it was not very modular, as it required writing a new module from the ground up if any of the implementation details were to be changed. Thus, in the spirit of modularity, we refactored the matrix library to implement an abstract `CONTAINER` signature. Then based on this signature, it became possible to create sub-signatures that specified the types of the CONTAINER signature. A good example is shown below:

```
signature CONTAINER = sig

  (* ADT of the container *)
  type 'a container
  (* type used to index the container *)
  type index
  (* type used to represent the size of the container *)
  type size
...
end

(* Matrices that are square and so only need size to be a single int *)
signature SQUAREMATRIX = sig

  include CONTAINER where type size = int

  val intToIndex : 'a container * int -> index

end
```

In this example we show the relevant parts of the `CONTAINER` signature and how we used that signature to create a *more specific* `SQUAREMATRIX` signature e.g. a matrix whose size datatype need only be a single integer value. This allows for a very flexible and modular implementation, in addition to square matrices we could create jagged two-dimensional matrices by following the same pattern viz. defining a `JAGGEDMATRIX` and setting size to "int * int".

Not only is the second layer of abstraction useful for changing matrix properties on the fly, but it is also useful for changing the underlying implementation and dimensionality of the matrix. With this design we were able to create a functor that, when given a structure of type `SQUAREMATRIX`, and a structure of type `VECT` will return a matrix of

whose underlying datatype is implemented by a structure, V of type `VECT`, that dictates the underlying datatype of the matrix, and a structure M, of type `SQUAREMATRIX` that specifies the datatype that stores the V structure. For example, we create matrices that are created with arrays, vectors and lists by passing each as a structure of type `VECT`; any such data structure is viable as long as it supports the `VECT` signature. By altering the structure M, of type `SQUAREMATRIX` we create n-dimensional matrices. So by varying both we gain the ability to create n-dimensional matrices, with abstracted underlying data types. Thus, the double abstracted design of our matrix library is a semantic definition of a matrix, which separates and abstracts the underlying datatype implementation from the dimensionality of the matrix and the underlying implementation details, thereby allowing for extreme reuse and flexibility.

3.5 The Functor is love, the Functor is life