N-dimensional Tic Tac Toe, an 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

The design for our game engine stemmed from trying to isolate the common general components of what would be required to create a running game. Rather than focus on creating one specific game implementation where these problems are addressed in an *ad hoc* way, we wanted to look at the larger picture and view the problem in terms of the common pattern that emerged between these kinds of games.

Each of the portions of the *State-Actions-Agents* design is independent so that - once created - it does not rely on other modules for its internal pieces. Each of the constituent parts of the game engine is more akin to a node on a graph and the interfaces are like edges. The pieces should be thought of as islands of code that could then interact with other modules. By using the game model to define a specific game - we essentially create more specific versions of those islands following the recipe spelled out in the signatures.

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 then its import into every functor would quickly become boilerplate and unwieldy. 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 programmer, seems to be encouraged by SML's module system, and is discussed in detail below.

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 a `CONTAINER` signature that specifies the interface for a container abstract data type. As above, we were able to use structural matching and the `include` keyword to extend our container ADT to more specific kinds of container. 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. Similarly, we can define a VECT signature that describes containers that can be used as one-dimensional vectors (like list or arrays) by restricting both the size and index types to be int. 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".

One consequence of this design is that you can create n-dimensional square matrices given only modules implementing VECT and a functor. We define a functor SquareMatrixFn that takes a module M implementing SQUAREMATRIX and a module V implementing VECT and produces a new square matrix of dimension n+1 where n is the dimensionality of M. We simply use V to store the polymorphic matrix elements and M to store rows of V. The type of indices of our new matrix is just the product of the index types of its components.

Because all VECT modules also implement SQUAREMATRIX, we can create two-dimensional matrices by passing any two VECT implementations to our functor. From there, we can create n-dimensional matrices by simply passing the result back into the functor as M any number of times. Therefore, given only a module implementing VECT, our functor can automatically produce modules representing matrices of n-dimensions. Because functors are not first-class, we cannot automate this process by passing in the intended dimensionality as an integer argument, so instead we are stuck threading the results of our functor back into itself. This could prove tedious for matrices with high dimensionality, but for our purposes was acceptable.

This use of functors also allows for mixing of containers in matrices. We implemented VECT for the basis array, list, and vector libraries. Our functor can take any combination of these and produce a matrix module that works the same as any other. As these implementation details are hidden by the container ADT, we can safely mix any of them to form another valid container matrix.

All of this has the result that our *Tic-tac-toe* implementation works for matrices of any dimensionality and implemented using a variety of data structures.

3.5 Coupling

One area where our implementation encountered some issues was in the coupling of our system. Despite our best efforts, our modules are not as loosely coupled as we would like. Each module carries with it the modules that it depends on as part of its structure. This is because many of the interfaces depend on type definitions contained in the modules that they depend on (for example, ACTION modules depend on the definition of the state type defined in STATE). This is not ideal in terms of coupling, and forced us to use a lot of trivial functors that thread these dependencies through our system.

Resolving this is a tricky problem. If we make the types of our abstract game interface all concrete, we could gain looser coupling at the cost of how generic our abstract game engine can be. Concrete type definitions in all signatures would alleviate this problem, but we would wind up unable to extend our abstract interface to any other games aside from *Tic-tac-toe*. Being locked into a single implementation in this way would simply reduce our modules to simple namespaces with none of the abstraction or extensibility benefits associated with modular programming.

One possible solution to this problem could be to somehow serialize interactions between different modules into a standard format (much like how web servers and clients often convert data into JSON to pass back and forth). However, this still seems to include some notion of coupling, just deferred now to serialization and deserialization processes. A module would still have to know how to serialize and deserialize information and how to interpret the communications it receives from other modules. So while we would decouple at the type level, we would still retain coupling at the informational level. It is unclear if there is a cleaner decomposition of our game domain that would allow for looser coupling while still retaining similar abstraction and cohesion as our current system.