The main model I used for the game took inspiration from the FRP Asteroids, Model-View controller architecture.

Controller

Starting from the beginning, our entire game originates from a merge of observable streams, those being the controller input from which takes the arrow keys and a game clock in which runs every 10 milliseconds. Rather than creating an entire stream for each key stroke, I created a key binding function with a closure inside of it allowing for reusability among different functions. In my example the the key bindings function was used purely to create a new instance of Translation, to which each keystroke being up, down, left, right corresponding to a translation of a frog in those directions. This was done by filtering the stream of keystrokes to only accept the arrow keys and then mapping the remaining arrow keys to the corresponding Translations to be handled further down the line within the Observer Handler function. For our stream of intervals, these were simply mapped to instances of tick such that our game states could be consistently tracked. These new created streams can be merged into one stream, ensuring that relative order of the incoming streams is kept.

Model Data

From this stream of observables in combination with an initial game state, a reduction of state can be produced that changes depending on the type of transformed observable that comes through. Qualifying keystrokes will change the state of our game by update the position of the frog, and almost all other transformations of state come from the tick. The original unaltered state is composed of important details on the initialisation of state as well as the ground items within our game. Any multiple occurring ground items are initialised using a pure functional map over an array in which the mapping function is a curried function such that objects can be created while only being partially initialised. The functionality this provides is setting certain elements of our object, while not potentially having the undetermined parts of our objects that can be set at a later period when it is determined. As previously mentioned this initial state of our game is transformed over the entirety of the game, but It is important to note that though the word transformed is being used, at no point are any states being manipulated, instead a constant stream of new states are being created, ensuring that all changes are localised to a specific function and no external changes are occurring throughout the code; a staple of functional reactive programming.

Following an instance of state being processed, a previous state is passed through our interactions functions that models our state based off the occurrences that are passively occurring throughout the intervals of our game. Within this function there are many higher order functions working to both map the arrays that are feature within the states, and filters being used to return boolean values responsible for the conditionals that make up our game state. Some of the arrays within our state serve similar responsibilities and hence can identical functions to map elements of the arrays, therefore to reduce our code into a more compact form we can concatenate these arrays into one and then use the map function on it. Note that all of this occurring as a result of an original reduction of our initial state from a stream of observables. This is done through the scan function of the subscription pipe.

Update View

The last step of the subscription pipe is the function that we subscribe to; updateView. Within this function is where we will have to do some imperative coding. As we are working with an SVG canvas, we must modify this as using a functional approach within this would just leading to repeated overlays of images on our SVG, hence we update rather than creating completely new copies every time a modification occurs. However within this function we can still use some examples of functional programming to reduce and stream line our code. For example when creating and updating our SVG a higher order forEach function is used to traverse over an object returning it's key and the value of its key. This can then be used to then set the attribute for the specific SVGs. Within this function, we also check conditionals to whether the game has ended from either a win or by the game ended from going out of bounds or other game ending reason. It can be noted that this is the only point within the entire file where an if statement occurs. If a

game has ended, we unsubscribe from the original game stream and display our game over messqage. If the game is a win however, we need to completely reset the SVG canvas which is done by manually deleting all dynamic elements of the SVG.