Arjun Biddanda (aab227)
Muhammad Khan (mhk98)
CS 3110 - Fall 2012
2:30 PHL Discussion
11/30/12

Problem Set 6

Part 7.4: PokeJouki Documentation

Summary

To implement *PokeJouki*, our first main task was to define the type `game`. Complying with our intuition, we thought of a game as mainly playing the role of a container of two teams. To follow through with this interpretation, we defined our own record type `team` containing its color, its inventory and its list of `cNUM_PICKS` steammon. A type `game` was subsequently defined as a 2-tuple where both types are `team ref`. We chose to use references to teams so that they could be modified later, which would have been very difficult if not impossible using a purely functional paradigm, although the functional style constitutes the style of the majority of the code. Such a definition also facilitated the implementations of the `game_datafication` and `game_from_data` functions in `game.ml` as it later proved facile to decompose each team into a pair of steammon list and attack list.

Game Design

To begin an actual implementation of the gameplay, we found it best to first modularize the situation as perfectly as possible, and so we created several `.ml` files such as `state.ml`, `item.ml` and `attack.ml`, among others. The purpose of each of these modules is to provide information/perform very localized calculations relevant to the portion of the game to which it corresponds. To give an example, `state.ml` contains functions that alter the state of the steammon, e.g. losing hp, losing pp and other things. The two main methods associated with any standard *PokeJouki* game are `init_game` and `handle_step`, which, respectively, start a new game and then process each step of the game along the way. In the former, we first begin by parsing the text files to obtain a list of attack and a pool of steammon from which to choose from. We parse by matching regular expressions via functions found in the OCaml `Str` module (not to be confused with the `String` module). After the steammon list is generated, a global reference variable pool is set to this steammon list so that this original pool could be accessed at any time. Lastly, default teams are created with zero steammon and zero items, to be filled in later on.

The $\texttt{handle}_{\texttt{s}}\texttt{tep}$ function processes each timestep of the game after teams have been filled and the battle is ready to begin. We defined a separate helper function `update_team` inside this function which takes in as arguments the states of the two teams and outputs a new team corresponding to the particular action that was taken during this timestep. The utility of defining a game using references to teams instead of just teams by themselves is very impressive, and is shown most of all in this function; without a reference, we would never be able to point to the actual location of the steammon/attack/item we want in memory. This function takes in a command and, as per the guidelines, only does something if this command is of type `Action`. Within a match statement, we match it against the several different actions and proceed accordingly. Conditionally, this function only sends `PickRequest` until each team has `cNUM_PICKS` steammon. This request becomes a `PickInventoryRequest` to get items, a `StarterRequest` to actually choose the steammon to send to battle, and lastly, once everything has been taken care of, an `ActionRequest` is sent which helps decides which attacks to perform or items to use against the opponent. All of these requests are handled by the bot, which signifies the artificial intelligence in this project.

Bot

In constructing the bot, there were three particular phases that were needed. The first phase was the draft phase in which we pick the steammon that we want on our team.The main theme behind our team was to do as much damage as possible to the opposing team. Therefore in our picking the steammon, we look at the most recent steammon that the opponent has picked and pick a powerful on that has a distinct type advantage against it. This "peeking" is available to us because of the structure of the game data. The way that a "powerful" pokemon is determined through a points function which assigns each steammon points based on the stats we valued. Then the type advantage is factored, to further filter the list of potential candidates. Then after this, we select the one with the highest score left in the list, which results in an optimum pick.

The second phase is to attack. This is our offense so we sought to hit hard and hit often. However, the type advantage helps greatly in generating a super-effective attack. So we have another scoring algorithm which gives each attack a score based on how effective it can be. Again we filter out the most powerful one (the highest score) and use it. This is the attack sequence at its most raw, effects are considered a side effect, not something that we are chasing after.

The final phase dually concerns the usage of items and the switching of steammon.Firstly, we must switch steammon when our current steammon is unable to battle. We utilize the same algorithms from our picking sequence, except we are only allowed to choose from our own team.As usual type advantage takes precedence, so that we can get the best possible steammon out there. If a steammon is stuck in battle for more than 10 turns, it becomes switched with one of its teammates largely because if it is out in the field for 10 turns that probably means that it is stuck in a loop where it and its opponent are immobalized. The use of items is minimal and quite frankly,hard coded into the system. The only items of concern in our battle strategy are the revives and the max potions. We use these items when certain conditions are met. For instance, using a max potion is not as simple as applying it if the steammon is under half health. It must be below half-health and there must be a boolean ref that is true in order to use the potion.This prevents onesteammon from "hogging" all of the potions to itself. Revives are used when 2 of the steammon on the team have fainted, but this is not subject to a constraint because we need the additional steammon to survive.

Testing

The whole idea behind the testing of our project was in the manipulation of the babybot.ml that was given to us. Initally after finishing a step, like the `PickRequest` section of `handle_step`, we would go to the babybot.ml, comment out whatever sections we did not need and then test that one case. This works extremely well for understanding the general idea and getting the ode to follow the type-checks of the build scripts. That being said these are not beneficial for the smaller test cases

The smaller test cases include subtleties like the picking order ($1, 2, 2, 2...., 2, 2, 1$) and applying effects to the steammon. The GUI updates were also extremely helpful in debugging the commands sent to the GUI. It is also much nicer to see visually how everything is interacting within the frame.

Challenges

Due to the size of the problem set, there are a large volume of things to cover within the code. It is very hard to go over the readme with a fine tooth comb and fully understand all of it. That being said, the way the handout is sectioned provides a very nice framework for modularity in this assignment. The two largest challenges that we had were getting the drafting order correct, when we are initially choosing steammon, and updating the values accurately within the mutable team type.

For choosing the steammon, the solution we finally stumbled upon was reached by drawing out the index of each pick and realizing the the team which got first pick would always get picks such that the integer division of its pick number by 2 was an even number, whereas the integer division of every pick number on other team by 2 was odd, which made it very easy to simulate the rules in the writeup.

The updating of the statistics for each steammon was slightly more confusing. This is because we had to deal with the side effects created because we wanted to make it mutable. We were not updating the previous references that we had created, but rather created new references with the updated characteristics that just are now sitting in memory and not being passed on. After some searching, we realized that we need to pass on the updated version instead of just doing nothing with it.

Another large challenge was how to divide the work. Primarily Arjun Biddanda(aab227) worked on the bot and the choosing algorithms for the steammom, whereas Muhammad Khan's (mhk98)main codebase was in the game.ml file. The external modules such as the `attack.ml` and the `item.ml` were developed as a collaborative effort.

Known Problems

One of our problems is in `game.ml` when the first team is chosen. Although we use a random number (`Random.int2` will always be either 0 or 1) to determine whether the red or blue team goes first, it always ends up that the blue team gets to pick first. This obviously cannot be a problem with the random number generator, but we are still unable to find the source of this nuisance in our code.

Comments

We spent approximately 3 weeks on this project, with much larger portion of the time involved during the third week. The advice to finish the game early should have been more stressed, as well as encouragement to simply read the handout. It would have been more beneficial to every student if the writeup included descriptions of all the functions, as the comments in the `.mli` files were rather sparse for a project of this magnitude. The visual nature of the project was very nice, as it is cool to see your work in a graphical setting. That being said, it might have been interesting to provide the code for the GUI asI'm sure a lot of students were curious about the methods Java can be interfaced with OCaml. Simply providing the jar hid that implementation from us. Although we realize that this complies with the principles of abstraction and modularity, it would be nice to have seen the code, from a pedagogical standpoint.