`replay`

A language for simple repeated games

Proposal

Eric Bolton - edb2129

COMS W4115 - Programming Languages and Translators
Fall 2016 - CVN
September 28, 2016

1 Motivation

Game theory is treacherous for humans. Looks can be deceptive: even the simplest games are tricky to analyze, and it's easy to make silly mistakes. Repeated versions of games compound these problems, adding layers of complexity and confusion. A natural answer is to turn to a computer program for help. However, programming languages embraced by game theorists, such as `MATLAB`, `Python`, or `R`, do little to simplify the task at hand, as they weren't natively designed for these analyses.

`replay` is an imperative programming language that makes finitely and infinitely repeated games easier to represent and analyze. The syntax provides a clear and straightforward way to specify the various parameters of simple games: basic strategies, payoffs, as well as more complicated variants of strategies. Then, the language enables simple, yet powerful analyses on these games, such as determining the set of dominated strategies, or discovering whether a strategy profile is a Nash equilibrium. Finally, `replay` makes it possible to string simple games together, enabling analyses of more complicated games - including finitely and infinitely repeated games.

2 Background

A repeated game is a version of a simple game that is played more than once. Oftentimes, the optimal strategy in the repeated version of a game is radically different from the one in the base game. For example, in the Prisoner's Dilemma, optimal play dictates that the players do not cooperate. But in the repeated version of the game, depending on the number of repetitions and the value players attribute to their future payoffs, the threat of punishment could encourage collaboration.

Another feature of games is information. Players may know differing amounts about what others in the game have played. An example is the distinction between simultaneous games and sequential games: in a simultaneous game such as Rock-Paper-Scissors, Player 2 does not know what Player 1 has played when she makes her move. This is not the case in a sequential game like Chess.

Finally, while games are typically described by the payoffs yielded by pure strategies, players may choose not to play a pure strategy. That is, they may play different moves with different odds. This is important in games of imperfect information such as simultaneous games. For example, in Rock-Paper-Scissors, it's a bad idea to always play Rock; it is better to attribute a probability of $\frac{1}{3}$ to each move.

3 Sample code

3.1 A simple game

Rock Paper Scissors can be described using two statements.

```
strat Shape {Rock | Paper | Scissors};
Game g = sim Shape * Shape -> [[(0, 0); (-1, 1); (1, -1)];
                               [(1, -1); (0, 0); (-1, 1)];
                               [(-1, 1); (1, -1); (0, 0)]];
```

This sample contains objects of the two key types of `replay`: `Game` and `Strategy`. `g` is of type `Game`, while `Rock`, `Paper`, and `Scissors` are of type `Strategy`, a subtlety that is explained by the `strat` keyword (see below).
Other data types available in `replay` are integers (`int`), booleans (`bool`), and floating point numbers (`float`).

The sample also demonstrates `replay`'s support for tuples, which are here used to designate the payoffs for each outcome of the game.

The sample introduces two keywords:

- **`strat`**: Similar in function to an enum, this keyword helps designate a set of pure strategies. The strategy set `Shape` describes the set of basic strategies in a game of Rock, Paper, Scissors.
- **`sim`**: This keyword specifies that the players play simultaneously, an important distinction. Had this keyword been omitted, player 2 would have 27 available pure strategies rather than 3: each describing three different responses for each of player 1's potential moves.

It also demonstrates the use of parentheses `()`, brackets `[]`, and curly braces {}. Parentheses are reserved for tuples, functions, and arithmetic, while curly braces are reserved for block delimitation and strategy enumerations (using bars `|` to separate elements). Meanwhile, brackets define arrays (using semi-colons ; to separate elements), and can be nested to define arrays of arrays.

Finally, note that a `Game` can specified using the `->` constructor, which maps a set of strategy profiles to a set of payoffs. The "`Shape * Shape`" expression is a shorthand for the set of each combination of two strategies, such as `(Rock, Scissors)`. Had the **`sim`** keyword been omitted, the lefthand "`Shape`" would have been interpreted to mean the strategy set of the player who plays second. This shorthand essentially represents a Cartesian product of the elements in `Shape`. The `->` operator demands two requirements:

1. The arrays on each side of the operator must have the same size and dimension
2. The payoff tuples and strategy profiles must have the same size

3.2 Flow control and functions

`replay` supports basic flow control with `if...else` statements and `for`-loops. This is principally helpful for writing user-defined payoff functions, which can then be used to describe objects of type `Game`.

Here is the matching pennies game described in this manner:

```
strat Side {Heads | Tails};

function match(Side s1, Side s2) {
     if (s1 == s2) {
          return (1, -1);
     }
     else {
          return (-1, 1);
     }
}

Game g = sim Side * Side -> match();
```

Pairs generated from the left side of the `->` are passed as arguments to the payoff function `match()`.

`replay` supports the basic boolean operators `&&, ||, !`, as well as the basic arithmetic operators `+, -, *, /` and comparison operators `==, !=, <, >, <=, >=`. It also supports exponentiation with `**`.

3.3 Repeated games

Finally, we examine the crux of `replay`: infinitely repeated games.

```
/* Prisoner's Dilemma. C means collaboration, N means no collaboration */
strat Option {C | N};

Game pd = sim Option * Option ->
[[(-1,-1) (-3, 0)];
[( 0,-3) (-2,-2)]];

Game infrep = pd ** inf;
infrep.p[1].delta = 0.5; // Default discount factor is 1. Change to 0.5.
infrep.p[2].delta = 0.5;

Strategy grim = cycle:     C ** inf
                | N:       N ** inf;

Strategy[] p = [grim; grim];

// The following statement prints true: this is a Nash Equilibrium
print(expected(grim, p, infrep) >= expected(N ^ inf, p, infrep));
```

This sample introduces the two crucial keywords for describing infinite games and strategies: `inf` and `cycle`, as well as the built-in functions `print` and `expected`.

The expression ”`pd ** inf`” indicates that the game `pd` is to be repeated infinitely. Likewise, `C ** inf` indicates that the strategy `N` is to be played for infinite periods. In both cases, `inf` could be replaced by a finite integer to indicate finite repetitions.

The keyword ”`cycle`” indicates that the strategy involves a cycle of a certain number of periods. For it to be applicable to a game, this cycle must not only consist of available moves, but it must also fully describe the player's strategy. In the case of repeated games, this means that it must have a strategy for each repetition of the game.

The ”`cycle: N ** inf | C: C ** inf;`” construction enables strategies to be defined as responses to certain moves. In this case, the default behavior is to play `N` infinitely, but if the other player plays a `C` during one period, then this behavior will change to playing `C` infinitely. For example, a tit-for-tat strategy would be defined by ”`cycle: N ** inf | C: C ** inf | N: N ** inf;`”

Finally, `replay` enables printing with the built-in functions `print` and `println`, and enables the calculation of the expected payoff of a strategy given the opponent's strategy profile using `expected`.