`replay`

Language Reference Manual

Eric Bolton - edb2129

Contents

1 Introduction **2**

2 Lexical conventions **2**

2.1 Tokens 2

2.2 Separators 3

3 Expressions **3**

3.1 Operations 3

3.2 Literals and identifiers 5

3.3 Non-primitive types 5

3.4 Attributes 7

4 Declarations **8**

4.1 Functions 8

4.2 Variables 9

4.3 `strat` enumerator 9

5 Statements **9**

5.1 Variable or strat declaration 9

5.2 Play, Mutate, and Crossover 10

5.3 Assignment 10

5.4 Return 10

5.5 If, else 10

5.6 For, in 11

5.7 List 11

6 Scope **11**

7 References **11**

1 Introduction

`replay` is an imperative programming language designed to make strategies in repeated games easier to represent and analyze. It draws inspiration from three papers in game theory, Abreu and Rubinstein (1988), Miller (1987), and Rubinstein (1986), which first formalized the use of automata theory for the analysis of games, an idea that was first suggested by Aumann (1981). As such, `replay` provides a framework for defining strategies as finite automata (Moore machines). In addition, it enables game payoffs to be specified functionally, simplifying the process of defining complex games. Finally, it provides tools central to the genetic algorithm introduced by Holland (1975), which has proven instrumental in the analysis of strategies in repeated games.

2 Lexical conventions

2.1 Tokens

There are five different kinds of tokens in `replay` : identifiers, keywords, literals, operators, and punctuation.

2.1.1 Variables

Variables begin with a letter, followed by any number of letters, digits, or underscores '_'.
The underscore '_' by itself denotes a wildcard in the context of moves (see 3. Expressions).

2.1.2 Keywords

The following identifiers are reserved for use as keywords:

`int`	`if`
`bool`	`else`
`float`	`for`
`string`	`in`
`void`	`return`
`Game`	`true`
`Strategy`	`false`
`Player`	`strat`
	`rand`

2.1.3 Literals

There are literals for each type, as follows:

- `int`: A sequence of digits.
- `bool`: A '`true`' or a '`false`'
- `float`: Two possibilities:
  - An integer part, a decimal point '`.`', a fraction part, and an exponent. The integer and fraction parts consist of a sequence of digits. The exponent consists of an `e`, followed by an optional sign `-` or `+`, and a sequence of digits. The integer and fraction parts are both optional, but at least one of the two must be present. Likewise, the decimal point and the exponent are both optional, but at least one must be present.

  - A '`rand`' keyword, which gets interpreted as a random floating point number between 0 and 1 at compile time.

- `string`: Any number of characters, delimited by quotes '`" "`'.

2.1.4 Operators

There are 18 operators, as follows:

```
+    ==    &&
-    !=    ||
*    >     !
/    <
=    >=
->   <=
%
#
~
```

2.1.5 Punctuation

The following characters are used as punctuation:

```
( )
[ ]
{ }
: ;
. ,
|
```

2.2 Separators

Comments and whitespace are ignored by the scanner, except to serve as separators.

- Comments are delimited by `/*` and `*/`.
- ' ' (space), '`\t`' (tab), '`\r`' (carriage return), and '`\n`' (newline) are treated as whitespace.

3 Expressions

3.1 Operations

An expression can be an operation, which consists of parentheses, expressions, unary operators, and binary operators.

3.1.1 Unary operators

Unary operators group right-to-left and have higher precedence than binary operators. They behave as follows:

- `-`*expression*: Unary negation. The result is the negative of the expression, without changing its type. It is applicable only to expressions of type `int` or `float`.
- `!`*expression*: Logical negation. The result is `true` if the expression is `false`. Conversely, it is `false` if the expression is `true`. It is applicable only to expressions of type `bool`.

3.1.2 Binary operators

Binary operators group left-to-right. Here they are, sorted from highest to lowest by precedence:

```
*   /
+   -
>   >=  <  <=
==  !=
&&
||
```

They behave as follows:

Arithmetic operators

- *expression*`*`*expression*: Multiplication. The result is `float` if one of the two operands is `float`. The result is `int` if both operands are `int`. If one operand is `float` and the other is `int`, the `int` gets converted to `float`. Multiplication is applicable only to expressions of type `int` or `float`.
- *expression*`/`*expression*: Division. Result and operand types behave the same way as with multiplication.
- *expression*`+`*expression*: Addition. Types behave the same way.
- *expression*`-`*expression*: Subtraction. Types behave the same way.

Relational operators

- *expression*`>`*expression*: Greater than.
- *expression*`<`*expression*: Less than.
- *expression*`>=`*expression*: Greater than or equal to.
- *expression*`<=`*expression*: Less than or equal to.

The result of these operators is `true` if the relation is true, and `false` if it is false. If one operand is `float` and the other is `int`, the `int` gets converted to `float`. Relational operators are applicable only to expressions of type `int` or `float`.

Equality operators

- *expression*`==`*expression*: Equal to.
- *expression*`!=`*expression*: Not equal to.

These behave the same as the relational operators, except that they are applicable to more types: any pair of operands that are of the same type. Additionally, if one operand is `float` and the other is `int`, the `int` gets converted to `float`.

Boolean operators

- *expression*`||`*expression*: Or. The result is `true` if either operand is `true`, and `false` otherwise. It is applicable only to operands of type `bool`.
- *expression*`&&`*expression*: And. The result is `true` if both operands are `true`, and `false` otherwise. Like `||`, it is applicable only to operands of type `bool`.

3.1.3 Parentheses

`(`*expression*`)`: The result is simply that of the expression enclosed in parentheses.

Note: special operators

`% # ~`: Play, cross, and mutate: Because these operators behave differently from the operators cited above, they are treated as statements. (See Section 5.2).

3.2 Literals and identifiers

3.2.1 Literals

An expression can be any of the literals specified in Section 2.1.3. The result is the type of the literal.

3.2.2 Identifiers

An expression can be an *identifier*, which in turn can be any of the following:

- *variable*: A variable, as specified in section 2.1.1.
- *variable*`(`*actuals*$_{opt}$`)`: The result of calling the function *variable* with parameters specified by the list of comma-separated values *actuals*$_{opt}$.
- *identifier*`[`*expression*`]`: The value at index *expression* of the array *identifier*. *expression* must be an `int`, while *expression* must be an array. (See Section 3.3 Non-primitive types)
- *variable*`.`*identifier*: The attribute *identifier* of *variable*. (See Section 3.4 Attributes)

3.3 Non-primitive types

An expression can also be a `Strategy`, a `Game`, a `Player`, or an array, the non-primitive types of `replay`.

3.3.1 Strategies

An expression can be:

- `Strategy[`*params*`]`: A Strategy. *params* is a list of *expressions*s separated by `|`'s. In this case, it must consist of the number of states (an `int`), the number of moves the strategy is based on (an `int`), and a list of *states* (see below). The numbers are needed to gauge memory requirements. This expression has type `Strategy`.
- {*states*}: A list of *state*'s, enclosed by braces and separated by semicolons. This is the last *expression* required by the `Strategy[`*params*`]` constructor.

States

A *state* is specified as follows:

variable`:`*expression*`,`*transitions*

variable is the name of the state, and is set to be an `int`. *expression* corresponds to the move the state outputs, which must be of type `int`. *transitions* is a list of transitions, separated by bars.

Transitions

A *transition* is specified as follows:

`(`*moves*`)->`*variable*

moves is a comma-separated list of either `int`'s or wildcards '_'. *variable* corresponds to the name of the state, and must be an `int`.
To summarize, here is an example Strategy expression:

```
Strategy[10 | 2 | {
  cooperate:  C, (_,D) -> defect
               | (_,C) -> cooperate;
  defect:     D, (_,_) -> defect; }]
```

This specifies a strategy with 10 states, and 2 possible moves. States don't need to have every possible transition specified; a state will by default map back to itself. Furthermore, not all states need to be specified: states will by default output whatever move is denoted by 0, and map back to themselves.

3.3.2 Games

An expression can be:

- `Game[`*params*`]`: A Game. In this case, *params* must consist of the number of moves (an `int`) in the game and a list of *outcomes* (see below). This expression has type `Game`.
- {*outcomes*}: A list of *outcome*'s, enclosed by braces and separated by bars `|`.

Outcomes

An *outcome* is specified as follows:

`(`*moves*`)->(`*payoffs*`)`

payoffs is a comma-separated list of *expression*'s, which must be of type `int`.
To summarize, here is an example Game expression:

```
Game[2 | { (C,C) -> (-1,-1)
| (C,D) -> (-5,0)
| (D,C) -> (0,-5)
| (D,D) -> (-3,-3); }]
```

A game's payoffs can also be specified functionally:

```
Game[2 | { (x,y) -> (x * y - 1, x * y + 1)}]
```

3.3.3 Players

An expression can also be a Player, specified as follows:

`Player[`*params*`]`

In this case, the first element of *params* must be a `Strategy`, corresponding to the Player's strategy. Then, the user can optionally specify additional parameters of type `float`. These correspond, in order, to:

- The Player's delta: how much value the player attributes to payoffs acquired in future rounds.
- The Player's output noise: the probability that a move by the Player will be incorrectly reported.
- The Player's input noise: the probability that the Player will receive an incorrect report of played moves.

This expression has type `Player`.
A Player tracks the state it is in, its accumulated payoff, and how many times it has played. Each time it "plays" in a Game, it adds the payoff it received, multiplied by its delta raised to the power of how many times it has played.
To summarize, here is an example Player expression:

```
Player[grim | 0.5 | 0.01 | 0.01]
```

3.3.4 Arrays

Finally, an expression can be arrays, specified as follows:

- *type*`[`*params*`]`: A *type* array. When *params* has only one value of type `int`, this construction always specifies an array.
- `[`$expression_1$`:`$expression_2$`]`: An integer array of size $expression_2$ `-` $expression_1$, whose values range from $expression_1$ to $expression_2$. Both *expression*'s are required to be `int`'s.

3.4 Attributes

`replay`'s non-primitive types have built-in attributes:

- `string`:
  - `len`: The length of the string.
- Arrays:
  - `len`: The length of the array.
- `Strategy`:

  - `size`: The number of states in the Strategy. (`int`)
  - `moves`: The number of moves the Strategy can play. (`int`)
  - `term`: The number of terminal states in the Strategy. (`int`)
  - `inacc()`: Counts the number of inaccessible states in the Strategy. (`int`)
  - `change(`*actuals*$_{opt}$`)`: Changes the state at a specified address to a new one *actuals*$_{opt}$ must be a list consisting of one `int`, and a list of one state. (`void`)

- `Game`:
  - `players`: The number of players in the game. (`int`)
  - `moves`: The number of moves each player can play. (`int`)
  - `payoff(`*actuals*$_{opt}$`)`: Returns the payoff of a specified player when a specified profile is played. *actuals*$_{opt}$ must be a list of one `int` and one `int` array, the `int` specifying the player, and the array specifying the moves in the profile. (`int`)
  - `change(`*actuals*$_{opt}$`)`: Changes the payoffs resulting from a specified profile to a new specified set of payoffs. *actuals*$_{opt}$ must be a list of two `int[]` arrays, the first one specifying the profile, the second one specifying the payoffs. (`void`)

- `Player`:
  - `strategy`: The strategy of the Player. (`Strategy`)
  - `state`: The state of the Player's Strategy the Player is in. (`int`)
  - `rounds`: The number of rounds the Player has played. (`int`)
  - `output`: The output noise level of the Player. (`float`)
  - `input`: The input noise level of the Player. (`float`)
  - `delta`: The delta of the Player. (`float`)
  - `payoff`: The accumulated payoff of the Player. (`float`)
  - `reset()`: Resets the accumulated payoff and the number of moves to 0. (`void`)

Each non-primitive type, with the exception of `string`, has a `string()` function as an attribute, which provides a string representation for the type.
Users cannot specify new attributes, nor can they change their value.

4 Declarations

There are three types of declarations: function declarations, variable declarations, and the `strat` enumerator.

4.1 Functions

A function declaration is specified as follows:

type variable `(`*formals*$_{opt}$`)`$\{$*statements*$\}$

type and *variable* specify the type and name of the function. *formals*$_{opt}$ specify the arguments the function takes, just as *actuals*$_{opt}$ specify the arguments passed in a function call. Finally, *statements* is a list of any number of *statement*'s, corresponding to the body of the function. (See Section 5. Statements) A function's last statement must be `return` statement, whose return type must correspond to the type of the function.

4.1.1 Built-in functions

Certain function names are reserved. They serve the following two purposes:
Printing

- `print(`*actuals*$_{opt}$`)`: Takes one argument, converts it to a `string` using its type's `string()` attribute, then prints it out. (`void`)
- `println(`*actuals*$_{opt}$`)`: Does the same, then prints a newline character. (`void`)

Type-casting

- `tofloat(`*actuals*$_{opt}$`)`: Takes an argument of type `int`, and returns a `float` corresponding to that `int`. (`float`)
- `toint(`*actuals*$_{opt}$`)`: Takes an argument of type `float`, and returns an `int` corresponding to the floor of that `float`. (`int`)

4.2 Variables

A variable declaration can take two forms:

- *type variable* `=` *expression*: This declares a *variable* of type *type*, and initializes it with the value *expression*. The types of the *expression* and the *variable* must match.
- *type variable*: This declares a *variable* of type *type*, and initializes it to a default value.

4.2.1 Arrays

In the case of an array declaration, the *type* is followed by `[]`:
- *type*`[]`*variable* `=` *expression*
- *type*`[]`*variable*

4.3 `strat` enumerator

Finally, a user can declare a set of moves using the `strat` enumerator:

`strat` {*variables*}

variables is a comma-separated list of n *variable*'s, which get declared with type `int` by this construction. They are initialized with values 0 through n depending on their position in the list.

5 Statements

There are many types of statements. A program consists of a list of statements and function declarations, in any order.

5.1 Variable or strat declaration

A statement can be either a variable or `strat` declaration, followed by a semicolon '`;`'

5.2 Play, Mutate, and Crossover

A statement can be any of the operations on Players.

- *identifiers*`%`*expression*`;`: Play. *identifiers* is a comma-separated list of *identifier*'s, which must be `Player`'s. *expression* must be a `Game`. This operator pits the Players against each other for a round of the Game, updating their payoffs and Strategy states.
- *identifier*`~`*expression*`;`: Mutate. *identifier* must be a `Player`, and *expression* must be a `float`. This operation acts on the bit representation of the `Player`'s strategy, changing each bit with a probability specified by the `float`.
- *identifier*$_1$`#`*expression*`#`*identifier*$_2$`;`: Crossover. *identifier*$_1$ and *identifier*$_2$ must be `Player`'s, and *expression* must be a `float`. This operation acts on the bit representations of the `Player`'s strategies, crossing the two representations at the position indicated by the `float`.

Now for an illustration of the effect of the following crossover statement's effect: `p1 # 0.75 # p2;`
Let P_1 be the bit representation of `p1`'s strategy, and P_2 be the bit representation of `p2`'s strategy.
Say before crossover, we have:
$P_1 = 11111111$ and $P_2 = 00000000$
After crossover, we would have:
$P_1 = 11111100$ and $P_2 = 00000011$. The bits following the position $\frac{3}{4}$, or 0.75 of the way into the bit representation have been swapped.

5.3 Assignment

A statement can be an assignment statement.

identifier`=`*expression*`;`

The *identifier* must be an array entry or a *variable*; it cannot be an attribute or a function call. The *expression* must be of the same type as the *identifier*. The value of the *expression* gets assigned to the *identifier*.

5.4 Return

A statement can be a return statement.

`return` *expression*$_{opt}$`;`

If *expression*$_{opt}$ is left empty, this returns a `void`. Otherwise, this returns an expression of type *expression*$_{opt}$.

5.5 If, else

A statement can be an if-else statement.

- `if (`*expression*`)` *statement*: *statement* is executed if and only if *expression* evaluates to `true`. *expression* must be a `bool`.
- `if (`*expression*`)` *statement*$_1$ `else` *statement*$_2$: If *expression* evaluates to `false`, *statement*$_2$ is executed.

5.6 For, in

A statement can be a for-loop.

`for` *variable* `in` *expression statement*

expression must be an array. For each iteration of the for-loop, *statement* is executed and *variable* takes on the next value of the array, until the end of the array is reached.

5.7 List

Finally, a statement can be a brace-enclosed list of statements:

{*statements*}

6 Scope

A *variable*'s scope reaches any line following their declaration. If they are declared within braces {}, they have local scope and cannot be reached outside of the braces.
If a *variable* is declared as part of a for-loop statement, its scope is limited to that for-loop's statement.
If an undeclared *variable* is used in a set of moves to define payoffs functionally within a list of *outcomes*, its scope reaches only the payoffs that directly follow the arrow of the *outcome* it is used in.
Finally, when a *variable* is used to name a state in a Strategy, its scope extends to all states in the strategy, whether they precede or follow it.

7 References

Abreu, D. and Rubinstein, A. "The Structure of Nash Equilibrium in Repeated Games with Finite Automata." *Econometrica* 56 (November, 1988):1259-81.
Aumann, R. T. "Survey of Repeated Games." In R. T. Aumann et al., eds., *Essays in Game Theory and Mathematical Economics in Honor of Oscar Morgenstern* (Bibliographisches Institut, Zurich: Mannheim), 1981:11-42.
Holland, T. H. *Adaptation in Natural and Artificial Systems.* Ann Arbor, Michigan: The University of Michigan Press, 1975.
Ritchie, Dennis M. *C Reference Manual* Bell Telephone Laboratories. 1978
Rubinstein, A. "Finite Automata Play the Repeated Prisoner's Dilemma." *JournalÂůof Eco- nomic Theory* 39 (June, 1986):83-96.
Miller, J. H. "The Coevolution of Automata in the Repeated Prisoner's Dilemma" SFI Working Paper: 1989–003