Università di Camerino

Poker Texas Hold'em using Maude

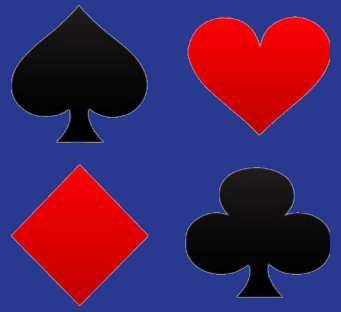
Antinori Alessandro
Coltrinari Riccardo
Scarpetta Marco

Fundamentals of Reactive Systems
A.Y. 2020-2021

Texas Hold'em in brief

One of the most popular variants of the card game of poker.

- Two cards are distributed face down to each player
- Five cards are distributed face up in three stages (flop, turn, river)
- Each player seeks the best five cards from any combination of the seven cards
- Players have betting options to check, call, raise, or fold
- Rounds of betting take place before the flop is dealt and after each stage
- The player who has the best hand (and has not folded) wins

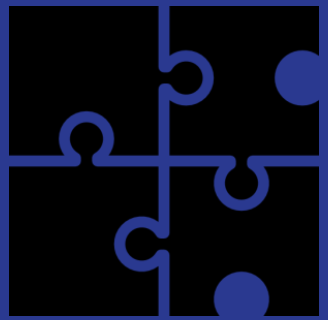


Goal

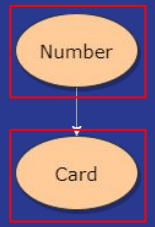
- ➔ How to design this poker game in Maude?
- ➔ How can we represent a player's hand to determine the winner?
- ➔ How can we simulate a game between two players?
- ➔ How can we use the rewriting rule for the betting phase?

Design

- ➔ The project is divided into several functional modules.
- ➔ It allows for a separation of responsibilities and easier reading.
- ➔ Bottom-Up approach (from the simple Card module to the Poker system module).



Design - Modules Relation



Card Number Module

Designed to represent the number of a card (from 2 to A).

The equal, greater than and less than operators have been redefined in order to compare two cards number.

There is also an operator to obtain a Nat from a Number (K = 13).

```
ops A 2 3 4 5 6 7 8 9 10 J Q K : -> Number [ctor] .
op ToNat : Number -> Nat .
op _equals_ : Number Number -> Bool .
op _>_ : Number Number -> Bool .
op _<_ : Number Number -> Bool .
```

```
Maude> red K .
reduce in POKER : K .
rewrites: 0 in 0ms cpu (0ms real)
result Number: K
Maude> red ToNat(A) .
reduce in POKER : ToNat(A) .
rewrites: 14 in 0ms cpu (0ms real)
result NzNat: 14
Maude> red J > 3 .
reduce in POKER : J > (3).Number .
rewrites: 16 in 0ms cpu (0ms real)
result Bool: true
```

Card Module

Designed to represent the card including the suit (clubs, diamonds, hearts, spades).

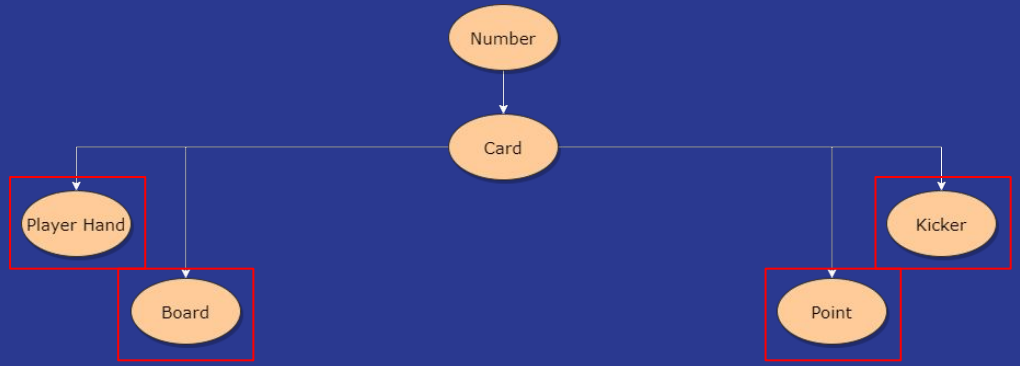
The equal, greater than and less than operators have been redefined in order to compare two cards.

This module has utility operators to get the number and the suit of a card.

```
ops C D H S : -> Suit [ctor] .
op [] : -> Card [ctor] .
op [__] : Number Suit -> Card [ctor] .
op CardNum : Card -> Number .
op CardSuit : Card -> Suit .
ops GetHigherCard GetLowerCard : Card Card -> Card .
```

```
Maude> red C .
reduce in POKER : C .
rewrites: 0 in 0ms cpu (0ms real)
result Suit: C
Maude> red [9 H] .
reduce in POKER : [9 H] .
rewrites: 0 in 0ms cpu (0ms real)
result Card: [9 H]
Maude> red [3 S] > [K D] .
reduce in POKER : [3 S] > [K D] .
rewrites: 24 in 0ms cpu (0ms real)
result Bool: false
```

Design - Modules Relation



Player Hand Module

Created to represent the two cards of a player .

The order relation operators have been redefined in order to compare two player hands.

There are also operators to get the card number or the suit of the two cards and operations to check for pair/flush.

```
op <> : -> PlayerHand [ctor] .
op <_,_> : Card Card -> PlayerHand [ctor] .
ops _equals_ _>_ _<_ : PlayerHand PlayerHand -> Bool .
op GetCard : PlayerHand Nat -> Card .
op GetSuitOfCard : PlayerHand Nat -> Suit .
op GetGreatestCard : PlayerHand -> Card .
op HasPairs : PlayerHand -> Bool .
op HasFlush : PlayerHand -> Bool .
```

```
Maude> red <[A C],[6 D]> .
reduce in POKER : < [A C],[6 D] > .
rewrites: 0 in 0ms cpu (0ms real) (~ rewri
result PlayerHand: < [A C],[6 D] >
Maude> red HasPairs(<[A C],[A D]>) .
reduce in POKER : HasPairs(< [A C],[A D] >
rewrites: 9 in 0ms cpu (0ms real) (~ rewri
result Bool: true
Maude> red <[A C],[4 D]> > <[9 H],[A H]> .
reduce in POKER : < [A C],[4 D] > > < [9 H
rewrites: 281 in 0ms cpu (0ms real) (~ rew
result Bool: false
```

Point Module

Represents the final point that a certain Player can have.

The main operators are the order relations (>, <, equals) and ToNat (assigns lower value for HCard, higher for Poker) .
They are useful to compare two Points and determine the winner.

There are also utility operators to get the card/suit representing the Point itself.

```
op HCard : -> Point [ctor] .
ops Pairs Tris Poker : Card -> Point [ctor] .
op DPairs : Card Card -> Point [ctor] .
op Straight : Nat -> Point [ctor] .
op Flush : Suit -> Point [ctor] .
op Full : Point Point -> Point [ctor] .
op GetCardOf : Point -> Card .
op GetSuitOf : Point -> Suit .
ops GetFirstCardOf GetSecondCardOf : Point -> Card .
op ToNat : Point -> Nat .
ops _>_ _<_ _equals_ : Point Point -> Bool .
```

```
Maude> red Tris([J H]) .
reduce in POKER-GAME-BOARD : Tris([J
rewrites: 0 in 0ms cpu (0ms real) (~
result Point: Tris([J H])
Maude> red GetSuitOf(Flush(H)) .
reduce in POKER-GAME-BOARD : GetSuitO
rewrites: 1 in 0ms cpu (0ms real) (~
result Suit: H
Maude> red Flush(H) > Poker([2 H]) .
reduce in POKER-GAME-BOARD : Flush(H)
rewrites: 4 in 0ms cpu (0ms real) (~
result Bool: false
```

Board Module

Created to represent the board of the game (five cards).

```
op _ _ _ _ _ : Card Card Card Card Card -> Board [ctor] .
ops HasCard HasPairsOf HasTrisOf HasFlush : Board Card -> Bool .
ops HasPairs HasDPairs HasTris HasFull HasPoker : Board -> Bool .
op GetCard : Board Nat -> Card .
op GetPairs : Board -> Card .
ops GetDPairs GetTris GetFull GetPoker : Board -> Point .
```

There are operators to check if a certain Point is present on the board and to retrieve the card(s) representing the Point itself.

```
ceq HasPoker(X Y Z V W) = true if HasTrisOf([] Y Z V W, X) .
ceq HasPoker(X Y Z V W) = true if HasTrisOf(X [] Z V W, Y) .
ceq HasPoker(X Y Z V W) = true if HasTrisOf(X Y [] V W, Z) .
ceq HasPoker(X Y Z V W) = true if HasTrisOf(X Y Z [] W, V) .
ceq HasPoker(X Y Z V W) = true if HasTrisOf(X Y Z V [], W) .
eq HasPoker(X Y Z V W) = false [owise] .
```

```
Maude> red in POKER-GAME-BOARD : [A C] [3 H] [A S] [4 D] [J C] .
reduce in POKER-GAME-BOARD : [A C] [3 H] [A S] [4 D] [J C] .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Board: [A C] [3 H] [A S] [4 D] [J C]
Maude> red HasPoker([A C] [A H] [A S] [4 D] [A D]) .
reduce in POKER-GAME-BOARD : HasPoker([A C] [A H] [A S] [4 D] [A D])
rewrites: 54 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
Maude> red GetTris([2 C] [A H] [A S] [4 D] [A D]) .
reduce in POKER-GAME-BOARD : GetTris([2 C] [A H] [A S] [4 D] [A D])
rewrites: 68 in 0ms cpu (0ms real) (~ rewrites/second)
result Point: Tris([A H])
```

Kicker Module

Represents the kicker that a certain Player can have, based on the point scored.

The kicker is composed by those cards, from the Player Hand and the Board, that are not part of the Point scored.

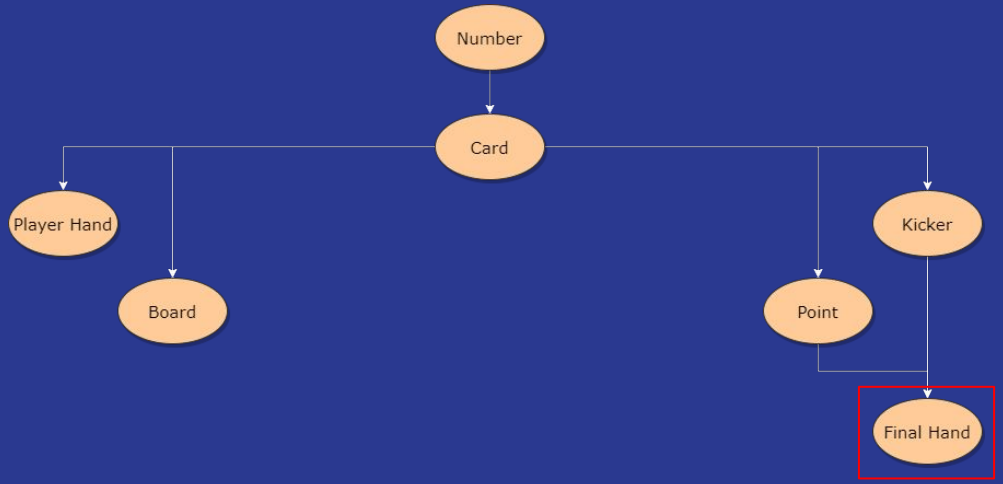
```
subsort List{Nat<} < Kicker .
op GetKicker : NeList{Nat<} -> Kicker [ctor] .
eq GetKicker(L) = reverse(sort(L)) .
ops _k>_ _k<_ _p>_ _p<_ _dp>_ _dp<_ _t>_ _t<_ : Kicker Kicker -> Bool .
ops _p==_ _p==_ _dp==_ _dp==_ _t==_ _t==_ _k==_ : Kicker Kicker -> Bool .
ops GetHigher GetSecondHigher GetThirdHigher GetFourthHigher GetFifthHigher : Kicker -> Nat .
```

There are different order relations that permit to compare 2 Kickers depending on the scored Point and decide which one is higher.

The comparison will have to be done whenever 2 Points are equals.

```
Maude> red in POKER-KICKER : GetKicker(7 9 3 12) .
reduce in POKER-KICKER : GetKicker(7 9 3 12) .
rewrites: 39 in 0ms cpu (0ms real) (~ rewrites/second)
result NeList{Nat<}: 12 9 7 3
Maude> red in POKER-KICKER : GetKicker(7 9 3 12) t== GetKicker(8 9 3 12) .
reduce in POKER-KICKER : GetKicker(7 9 3 12) t== GetKicker(8 9 3 12) .
rewrites: 92 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
Maude> red in POKER-KICKER : GetSecondHigher(GetKicker(7 9 3 12)) .
reduce in POKER-KICKER : GetSecondHigher(GetKicker(7 9 3 12)) .
rewrites: 42 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: (9).NzNat
```

Design - Modules Relation



Final Hand Module

A Final Hand is composed by a Point and a Kicker.

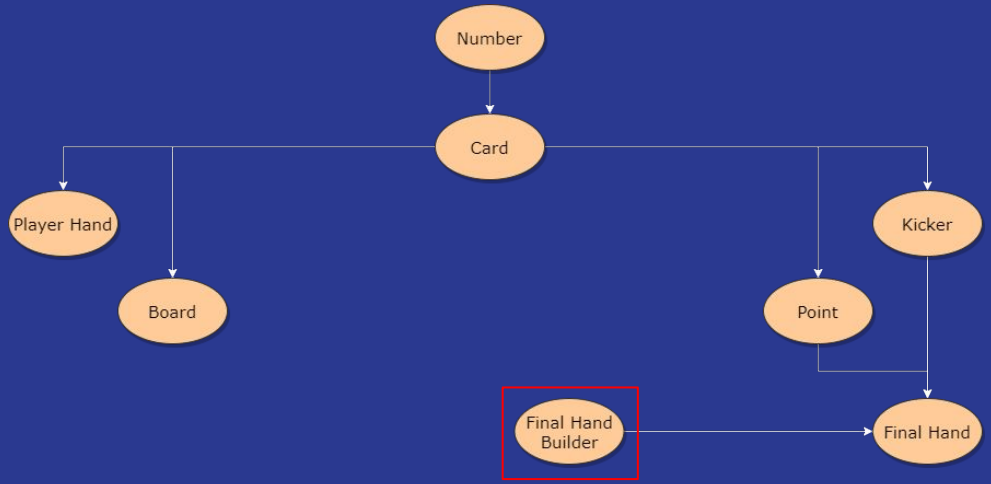
It is used to simplify the future operations that will permit to evaluate the best hand.

For each Player we will just have to calculate their Final Hands and then compare the latter to know which is the best, still using the > and < operators.

```
op _#_ : Point Kicker -> FinalHand [ctor] .
op GetPoint : FinalHand -> Point .
op GetKicker : FinalHand -> Kicker .
ops _>_ _<_ _equals_ : FinalHand FinalHand -> Bool .
```

```
Maude> red Pairs([3 C]) # GetKicker(4 8 7 12 6) .
reduce in POKER : Pairs([3 C]) # GetKicker((4).NzNat (8).NzNat (7).NzNat 12 (
    6).NzNat) .
rewrites: 56 in 0ms cpu (0ms real) (~ rewrites/second)
result FinalHand: Pairs([3 C]) # 12 8 7 6 4
Maude> red GetPoint(Pairs([3 C]) # GetKicker(4 8 7 12 6)) .
reduce in POKER : GetPoint(Pairs([3 C]) # GetKicker((4).NzNat (8).NzNat (
    7).NzNat 12 (6).NzNat)) .
rewrites: 57 in 0ms cpu (0ms real) (~ rewrites/second)
result Point: Pairs([3 C])
Maude> red Tris([3 C]) # GetKicker(4 8 7 12) > Pairs([4 H]) # GetKicker(8 7 12
3) .
reduce in POKER : Tris([3 C]) # GetKicker((4).NzNat (8).NzNat (7).NzNat 12) >
    Pairs([4 H]) # GetKicker((8).NzNat (7).NzNat 12 (3).NzNat) .
rewrites: 83 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
Maude> red Tris([3 C]) # GetKicker(4 14 7 12) > Tris([3 H]) # GetKicker(13 4 7
12 ) .
reduce in POKER : Tris([3 C]) # GetKicker((4).NzNat 14 (7).NzNat 12) > Tris([3
    H]) # GetKicker(13 (4).NzNat (7).NzNat 12) .
rewrites: 127 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Design - Modules Relation



Final Hand Builder Module

In order to create a Final Hand we need to take a Player Hand and a Board as input, and then calculate the Point scored and the Kicker.

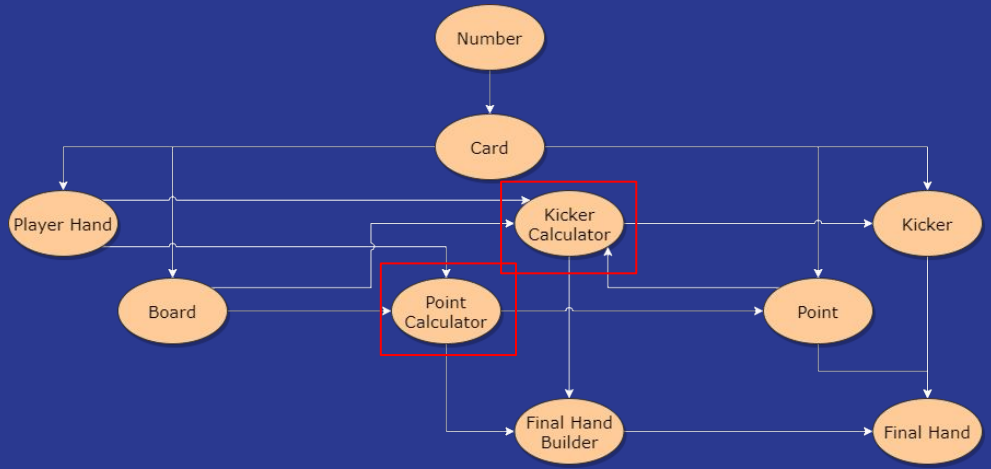
```
vars X Y : PlayerHand .
vars Z V : Board .
op GetFHand : PlayerHand Board -> FinalHand .
eq GetFHand(X, Z) = EvaluateHand(X, Z) # GetKicker(X, Z, EvaluateHand(X, Z)) .
```

To do that it makes use of 2 modules that help making calculations:

- Point Calculator
- Kicker Calculator

```
Maude> red GetFHand(<[A C],[6 D]>, [A C] [3 H] [A S] [4 D] [J C]) .
reduce in POKER : GetFHand(< [A C],[6 D] >, [A C] [3 H] [A S] [4 D]
rewrites: 5536 in 0ms cpu (2ms real) (~ rewrites/second)
result FinalHand: Tris([A C]) # 11 6 4 3
```

Design - Modules Relation



Point Calculator Module

Calculate the Point scored from a Player Hand and a Board.

It is defined to calculate the Point in a top-down manner, from the higher to the lowest possible Point.

We start to check if a Poker has been scored and, if not, we check the next Point (Full), until we reach the lowest possible Point (High Card).

```
ceq CheckPoker(X, Z) = Poker(GetCard(X, 1)) if HasTrisOf(Z, GetCard(X, 1)) .
ceq CheckPoker(X, Z) = Poker(GetCard(X, 2)) if HasTrisOf(Z, GetCard(X, 2)) .
ceq CheckPoker(X, Z) = Poker(GetCard(X, 1)) if HasPairs(X) and HasPairsOf(Z, GetCard(X, 1)) .
ceq CheckPoker(X, Z) = GetPoker(Z) if HasPoker(Z) .
eq CheckPoker(X, Z) = CheckFull(X, Z) [owise] .
```

```
Maude> red EvaluateHand(<[A C],[4 S]>, [A H] [3 S] [A S] [A D] [3 H]).
reduce in POKER : EvaluateHand(< [A C],[4 S] >, [A H] [3 S] [A S] [A D]
rewrites: 56 in 0ms cpu (0ms real) (~ rewrites/second)
result Point: Poker([A C])
```

```
op GetAsList : PlayerHand Board -> List{Nat<} .
eq GetAsList(X, Z) = sort( makeList( union(GetAsSet(X), GetAsSet(Z)) )  ).
op EvaluateHand : PlayerHand Board -> Point .
eq EvaluateHand(X, Z) = CheckPoker(X, Z) .
ops CheckPairs CheckDPairs CheckTris CheckStraight CheckFlush CheckFull CheckPoker : PlayerHand Board -> Point .
```

Kicker Calculator Module

Gets a Kicker from a Board, a Player Hand and the Point scored.

Both the Board and the Player Hand are pushed into a set in order to get rid of repeated cards (cards with same Number).

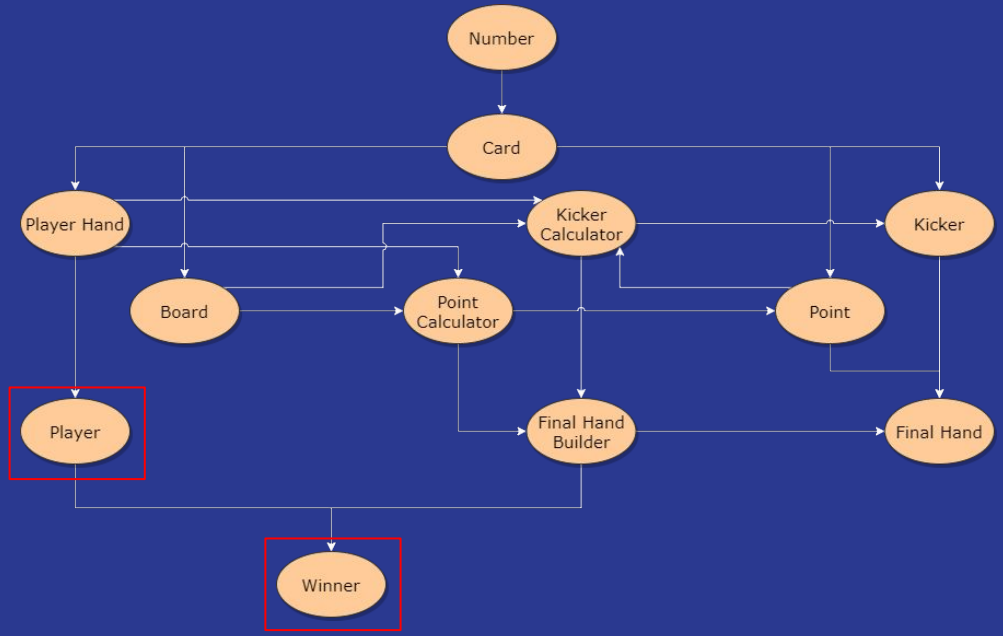
```
op GetAsSet : PlayerHand Board -> Set{Nat} .
op GetSetAsList : Set{Nat} -> List{Nat<} .
op DeletePointFromSet : Set{Nat} Point -> Set{Nat} .
op GetKicker : PlayerHand Board Point -> Kicker .
eq GetKicker(PH, PL, P) = GetKicker( GetSetAsList( DeletePointFromSet( GetAsSet(PH, PL), P) ) ) .
```

Then the cards that form the Point are discarded.

Finally the set is transformed into an ordered list from the highest to the lowest card.

```
Maude> red GetKicker(<[A C],[6 D]>, [6 C] [6 H] [K S] [Q D] [J C], Tris([6 D])) .
reduce in POKER : GetKicker(< [A C],[6 D] >, [6 C] [6 H] [K S] [Q D] [J C], Tris(
rewrites: 139 in 0ms cpu (0ms real) (~ rewrites/second)
result NeList{Nat<}: 14 13 12 11
```

Design - Modules Relation



Player Module

Created to represent two players, P1 and P2.

A Player simply consists of a PlayerHand.

```
ops P1 P2 : PlayerHand -> Player .
op GetPlayerId : Player -> Nat .
op GetPlayerHand : Player -> PlayerHand .
op HasPairs : Player -> Bool .
```

```
Maude> red P1(<[A C],[6 D]>) .
reduce in POKER : P1(< [A C],[6 D] >) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites
result Player: P1(< [A C],[6 D] >)
Maude> red GetPlayerId(P2(<[A C],[6 D]>)) .
reduce in POKER : GetPlayerId(P2(< [A C],[6 D
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites
result NzNat: (2).NzNat
Maude> red GetPlayerHand(P2(<[A C],[6 D]>)) .
reduce in POKER : GetPlayerHand(P2(< [A C],[6
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites
result PlayerHand: < [A C],[6 D] >
```

There are operations to get the Id or the PlayerHand of a Player and to check if it has a Pair.

Winner Module

Defines a Winner that could be either a Player or a Draw.

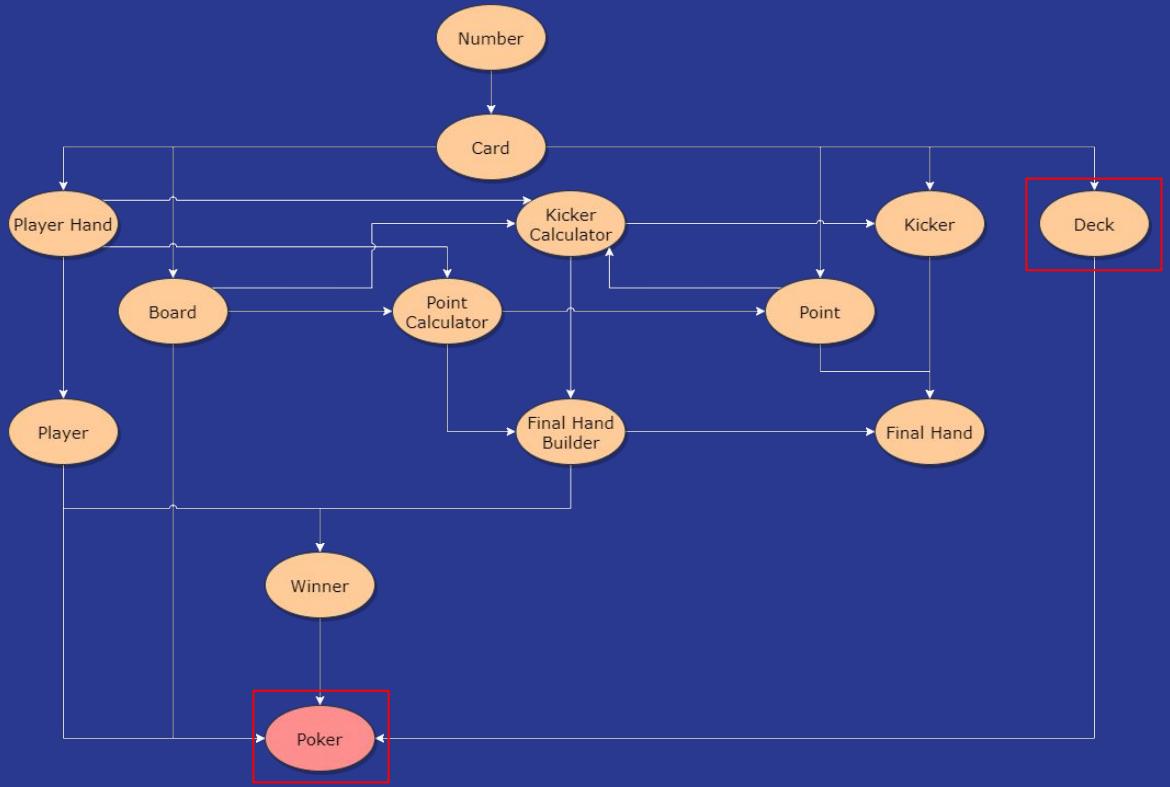
```
op _/_ : Player Point -> Winner [ctor] .
op Draw : -> Winner [ctor] .
op _vs_board_ : Player Player Board -> PokerGame [ctor] .
op Winner : PokerGame -> Winner .
ceq Winner(F vs L board P) = F / EvaluateHand(GetPlayerHand(F), P) if GetFHand(GetPlayerHand(F), P) > GetFHand(GetPlayerHand(L), P) .
ceq Winner(F vs L board P) = L / EvaluateHand(GetPlayerHand(L), P) if GetFHand(GetPlayerHand(F), P) < GetFHand(GetPlayerHand(L), P) .
eq Winner(F vs L board P) = Draw [owise] .
```

Here is also defined the function that given 2 Players and a Board tells which Player won the game or if its a Draw.

This is done by checking which one of the 2 Players has the best hand, using the > and < operators of the Final Hand.

```
Maude> red Winner(P1(<[10 H],[J S]>) vs P2(<[A S],[Q H]>) board [10 S] [Q D] [3 H] [A C] [10 D]) .
reduce in POKER : Winner(P1(< [10 H],[J S] >) vs P2(< [A S],[Q H] >) board [10 S] [Q D] [3 H] [A C
rewrites: 15219 in 4ms cpu (3ms real) (3804750 rewrites/second)
result Winner: P1(< [10 H],[J S] >) / Tris([10 H])
```

Design - Modules Relation



Deck Module (1/2)

This module was created to represent a deck of card constructed from cards by induction (_ _ operator).

The deck module thus created provides some useful operators to get information on the deck, such as the length, or to modify it, via the remove operator.

The remove operator was implemented for all possible cases (e.g. remove a card from an empty deck, from a deck with a single card, from a generic deck), since a card can be removed from every possible index.

```
op [] : -> Deck [ctor] .
op __ : Card Deck -> Deck [ctor] .
op tail : Deck -> Deck .
op head : Deck -> Deck .
op length : Deck -> Int .
```

```
op remove : Deck Int -> ReturnedDeckAndCard .
eq remove([], N) = {[], []} .
ceq remove(C, N) = {[], C} if N == 0 .
ceq remove(C, N) = {C, []} if N > 0 .
ceq remove(L, N) = {tail(L), head(L)} if N == 0 .
ceq remove(L, N) = {head(L), remove(tail(L), N - 1)} if N > 0 and length(tail(L)) > 1 .
ceq remove(L, N) = {L, []} if N > length(L) or N == length(L) .
ceq remove(L, N) = {head(L), tail(L)} if N == 1 and length(tail(L)) == 1 .
```

Deck Module (2/2)

For the previous reasons and for the fact that the deck has been constructed by induction, operators such as length or remove present some recursive definition.

As for the remove operator, the return type is a Maude sort that we called ReturnedDeckAndCard, that is able to retain information such as the new deck without a card and the removed card itself.

This let us draw a card from the deck, update the deck and put the removed card in a player's hand or on the board.

```
op length : Deck -> Int .
...
eq length(C L) = length(C) + length(L) .

op remove : Deck Int -> ReturnedDeckAndCard .
...
ceq remove(L, N) = {head(L), remove(tail(L), N - 1)} if N > 0 and length(tail(L)) > 1 .
```

Poker Module

For our project, we wrote a .maude file for each module and we created a poker.maude system module to load all the modules needed via the *load* keyword and import them via the *protecting* keyword.

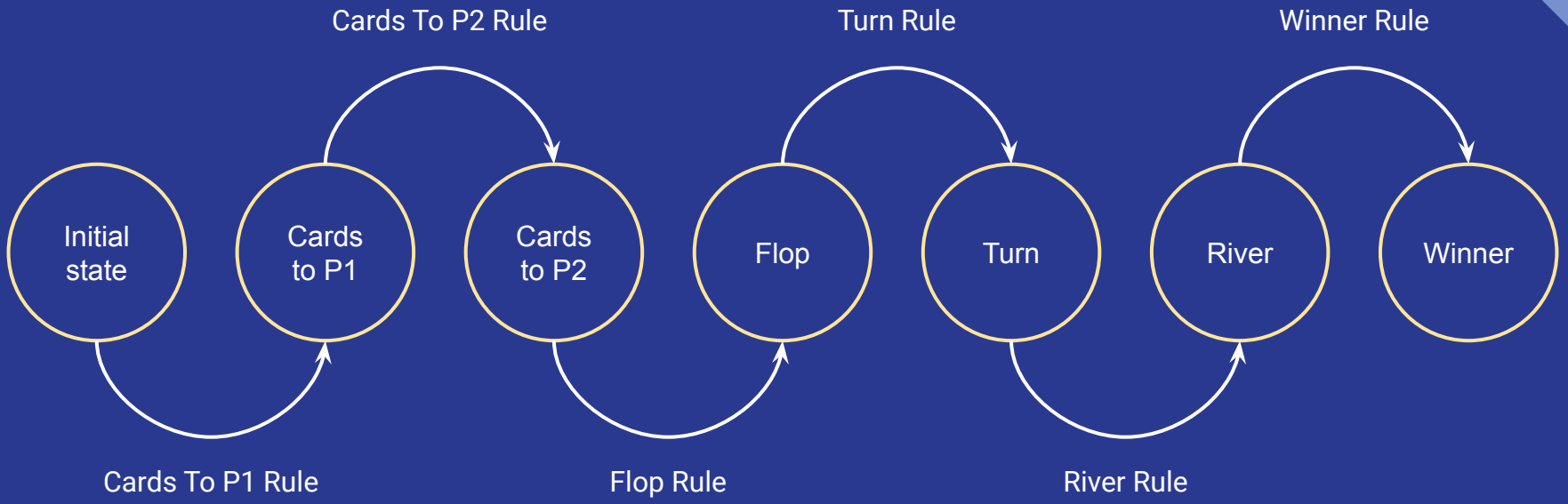
The poker.maude module also contains all the operators and rules needed to make the game evolve.

```
load poker-lists-and-sets
load poker-card-number
load poker-card
load poker-deck
load poker-player-hand
load poker-player
load poker-point
load poker-game-board
load poker-kicker
load poker-final-hand
load poker-points-calculator
load poker-kicker-calculator
load poker-final-hand-builder
load poker-winner

mod POKER is
    pr POKER-LISTS-AND-SETS .
    pr POKER-CARD-NUMBER .
    pr POKER-CARD .
    pr POKER-DECK .
    pr POKER-PLAYER-HAND .
    pr POKER-POINT .
    pr POKER-GAME-BOARD .
    pr POKER-KICKER .
    pr POKER-FINAL-HAND .
    pr POKER-POINTS-CALCULATOR .
    pr POKER-KICKER-CALCULATOR .
    pr POKER-FINAL-HAND-BUILDER .
    pr POKER-PLAYER .
    pr POKER-WINNER .
    pr RANDOM .
```

```
→  maude-3.1 git:(main) ✗ ./maude.linux64 poker.maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 3.1 built: Oct 12 2020 20:12:31
            Copyright 1997-2020 SRI International
                   Sun Jan 10 11:35:14 2021
Maude> red [2 C] [3 D] [4 H] .
reduce in POKER : [2 C] [3 D] [4 H] .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Deck: [2 C] [3 D] [4 H]
Maude> red random(10) .
reduce in POKER : random(10) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 2774094101
```

States and Rules



State and Rewriting Logic in Poker (1/2)

In Maude, unconditional (rl) and conditional rules (crl) make the system evolve, going from a state S_n to a state S_m such that S_n and S_m are terms of the same kind. That's why, in order to simulate a Poker round, we had to consider each step as a term belonging to the same sort.

```
sort State .
op ____ : Board Player Player Deck -> State [ctor] .
op _____ : Board Player Player Deck Nat -> State [ctor] .
op ______ : Board Player Player Deck Winner Nat -> State [ctor] .
```

In our case, during each step, the state of the game is given by the Board, the Players, the Deck and optionally by the Pot and the Winner.

State and Rewriting Logic in Poker (2/2)

We then proceeded to write all the needed operators, equations and conditional rules.

For each transition between states we defined something like this:

```
op cardsToP1 : State Nat -> State .
eq cardsToP1(B P1 P2 L, N) = ...
crl [cardsToP1Rule] : B P1 P2 L N => cardsToP1(B P1 P2 L, N) if GetPlayerHand(P1) == <> .
```

so that we are able to separate the equations that will change the variables that make up the state from the conditions that allow for these changes.

Random Module (1/2)

In order to remove a card randomly from the deck each time a card is drawn, we made use of the Random Maude module.

Doing things this way, we implemented the following draw operator:

```
op draw : Deck -> [ReturnedDeckAndCard] .
eq draw(D:Deck) = remove(D:Deck, random(length(D:Deck)) rem length(D:Deck)) .
```

Normally, calling random(x) with the same value of x generates the same number, but each time a card is drawn, it is removed from the deck, thus decreasing the deck's length and allowing for a different index of the card to draw each time we call random.

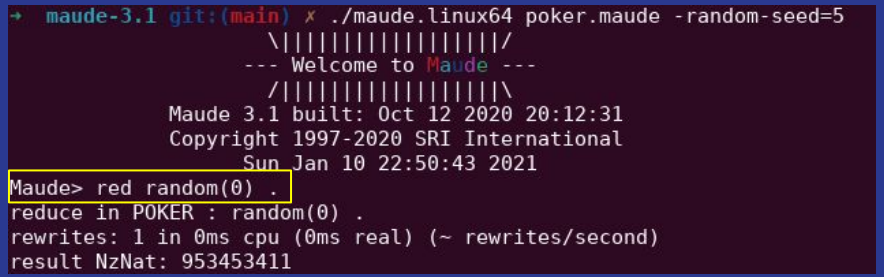
In addition to that, we had to take the modulo (rem operator) of the number generated by random, since it was in the range [0, 2^{32}-1] and we had to restrict it to the range [0, length of the deck].

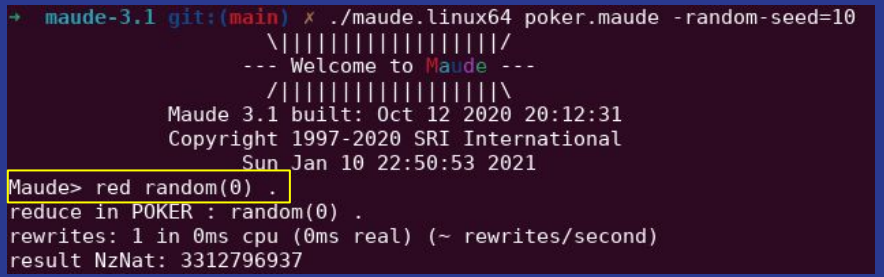
Random Module (2/2)

As we said, all the cards are drawn randomly from the deck, but given the same initial seed (0 in case it is not specified), the sequence of numbers generated by random are always the same.

The poker round can therefore end in only one manner.

One way to avoid this is to use the *-random-seed* command line option to use a different seed for the random function, in order to simulate different rounds.





Round example

Now that we have defined all the modules used, we can simulate a round of poker!

```
Maude> rew [] [] [] [] [] P1(<>) P2(<>) init 0 .
rewrite in POKER : [] [] [] [] [] P1(<>) P2(<>) init 0 .
rewrites: 80032 in 16ms cpu (17ms real) (5002000 rewrites/second)
result State: [3 D] [5 H] [5 S] [10 H] [9 H] P1(< [8 S],[J S] >) P2(< [3 S],[K
    S] >) [2 C] [3 C] [4 C] [5 C] [6 C] [7 C] [8 C] [9 C] [10 C] [J C] [Q C] [K
    C] [A C] [2 D] [4 D] [5 D] [6 D] [7 D] [8 D] [9 D] [10 D] [J D] [Q D] [K D]
    [A D] [2 H] [3 H] [4 H] [6 H] [7 H] [8 H] [J H] [Q H] [K H] [A H] [2 S] [4
    S] [6 S] [7 S] [9 S] [10 S] [Q S] [A S] P2(< [3 S],[K S] >) / DPairs([3 S],
    [5 H]) 30
```

```
Maude> search [] [] [] [] [] P1(<>) P2(<>) init 0 =>* B:Board P1:Player P2:Playe
r D:Deck W:Winner N:Nat such that GetWinner(W:Winner) == 2 and N > 50 .
search in POKER : [] [] [] [] [] P1(<>) P2(<>) init 0 =>* B P1 P2 D:Deck W N
    such that N > 50 and GetWinner(W) == (2).NzNat = true .

Solution 1 (state 21)
states: 22  rewrites: 309623 in 68ms cpu (67ms real) (4553279 rewrites/second)
B --> [3 D] [5 H] [5 S] [10 H] [9 H]
P1 --> P1(< [8 S],[J S] >)
P2 --> P2(< [3 S],[K S] >)
D:Deck --> [2 C] [3 C] [4 C] [5 C] [6 C] [7 C] [8 C] [9 C] [10 C] [J C] [Q C] [
    K C] [A C] [2 D] [4 D] [5 D] [6 D] [7 D] [8 D] [9 D] [10 D] [J D] [Q D] [K
    D] [A D] [2 H] [3 H] [4 H] [6 H] [7 H] [8 H] [J H] [Q H] [K H] [A H] [2 S]
    [4 S] [6 S] [7 S] [9 S] [10 S] [Q S] [A S]
W --> P2(< [3 S],[K S] >) / DPairs([3 S], [5 H])
N --> 60

Solution 2 (state 22)
states: 23  rewrites: 348125 in 76ms cpu (75ms real) (4580592 rewrites/second)
B --> [3 D] [5 H] [5 S] [10 H] [9 H]
P1 --> P1(< [8 S],[J S] >)
P2 --> P2(< [3 S],[K S] >)
D:Deck --> [2 C] [3 C] [4 C] [5 C] [6 C] [7 C] [8 C] [9 C] [10 C] [J C] [Q C] [
    K C] [A C] [2 D] [4 D] [5 D] [6 D] [7 D] [8 D] [9 D] [10 D] [J D] [Q D] [K
    D] [A D] [2 H] [3 H] [4 H] [6 H] [7 H] [8 H] [J H] [Q H] [K H] [A H] [2 S]
    [4 S] [6 S] [7 S] [9 S] [10 S] [Q S] [A S]
W --> P2(< [3 S],[K S] >) / DPairs([3 S], [5 H])
N --> 70

Solution 3 (state 23)
states: 24  rewrites: 386627 in 84ms cpu (82ms real) (4602702 rewrites/second)
```

That's All Folks!