Jogo da Velha em Quint

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC

16 de abril de 2025

Conteúdo

Outline

Jogo da Velha

Todos conhecem jogo da velha?

Jogo da Velha

Todos conhecem jogo da velha?

PS: a partir de hoje, não vou mais traduzir as especificações para português

- Acho legal usar português no início para que fique claro o que são `keywords` (sempre em inglês) e o que podemos escolher o nome (nesses casos, em português)
- A partir daqui, vou usar os exemplos originais, em inglês.

Tipos e variáveis

Definimos os seguintes tipos

```
type Player = X | O
type Square = Occupied(Player) | Empty
```

E as seguintes variáveis

```
/// A 3x3 tic-tac-toe board
var board: int -> int -> Square

/// Who goes next
var nextTurn: Player
```

Definições sobre coordenadas

```
1  pure val boardCoordinates = tuples(1.to(3), 1.to(3))
2
3  def square(coordinate: (int, int)): Square =
4    board.get(coordinate._1).get(coordinate._2)
5
6  def hasPlayer(coordinate, player) =
7    match square(coordinate) {
8      | Empty        => false
9      | Occupied(p) => player == p
10   }
11
12 def isEmpty(coordinate) =
13   match square(coordinate) {
14     | Empty  => true
15     | _      => false
16   }
```

Definições sobre o tabuleiro

```
1 val boardEmpty = boardCoordinates.forall(isEmpty)
2
3 val boardFull = not(boardCoordinates.exists(isEmpty))
```

Definindo “ganhar” - coordenadas

- Como o tabuleiro é sempre 3x3, é mais fácil listar todas as combinações de coordenadas que levam a uma vitória do que implementar os cálculos.
- Usamos `Set` - não precisamos de ordem nem de repetição, logo não devemos usar `List`.

```
pure val winningPatterns = Set(
  // Horizonal wins
  Set((1,1), (1,2), (1,3)),
  Set((2,1), (2,2), (2,3)),
  Set((3,1), (3,2), (3,3)),
  // Vertical wins
  Set((1,1), (2,1), (3,1)),
  Set((1,2), (2,2), (3,2)),
  Set((1,3), (2,3), (3,3)),
  // Diagonal wins
  Set((1,1), (2,2), (3,3)),
  Set((3,1), (2,2), (1,3))
)
```

Definindo “ganhar” - operador won

Usamos as definições para `winningPaterns` e `hasPlayer` para determinar se um jogador venceu.

```
1 def won(player) = winningPatterns.exists(pattern =>
2   pattern.forall(coordinate => hasPlayer(coordinate,
      player))
3 )
```

Definindo “ganhar” - operador won

Usamos as definições para `winningPaterns` e `hasPlayer` para determinar se um jogador venceu.

```
def won(player) = winningPatterns.exists(pattern =>
  pattern.forall(coordinate => hasPlayer(coordinate,
    player))
)
```

Com essa definição e `boardFull`, podemos determinar se um jogo já acabou.

```
val gameOver = won(X) or won(O) or boardFull
```

- Reparem que o operador `or` pode ser usado na forma infixa (no meio dos argumentos)

Ações - Move

Um dado jogador faz uma jogada (um *move*) em uma dada coordenada

- Determinístico

```
action Move(player, coordinate) = all {
  isEmpty(coordinate),
  board' = board.setBy(
    coordinate._1,
    row => row.set(coordinate._2, Occupied(player))
  ),
}
```

- Qual é a pré-condição pra essa ação?

Ações - Move

Um dado jogador faz uma jogada (um *move*) em uma dada coordenada

- Determinístico

```
action Move(player, coordinate) = all {
  isEmpty(coordinate),
  board' = board.setBy(
    coordinate._1,
    row => row.set(coordinate._2, Occupied(player))
  ),
}
```

- Qual é a pré-condição pra essa ação?
  - A pré-condição para essa ação é que a coordenada esteja vazia

Ações - Move

Um dado jogador faz uma jogada (um *move*) em uma dada coordenada

- Determinístico

```
action Move(player, coordinate) = all {
  isEmpty(coordinate),
  board' = board.setBy(
    coordinate._1,
    row => row.set(coordinate._2, Occupied(player))
  ),
}
```

- Qual é a pré-condição pra essa ação?
  - A pré-condição para essa ação é que a coordenada esteja vazia
- `setBy` é bem útil pra atualizar mapas aninhados (como nesse caso, `int -> int -> Square`)

Ações - MoveToEmpty

Um dado jogador faz uma jogada em **alguma** coordenada

- Não-determinístico

```
action MoveToEmpty(player) = all {
  not(gameOver),
  nondet coordinate = boardCoordinates.filter(isEmpty)
    .oneOf()
  Move(player, coordinate)
}
```

- Qual é a pré-condição pra essa ação?

Ações - MoveToEmpty

Um dado jogador faz uma jogada em **alguma** coordenada

- Não-determinístico

```
action MoveToEmpty(player) = all {
  not(gameOver),
  nondet coordinate = boardCoordinates.filter(isEmpty)
    .oneOf()
  Move(player, coordinate)
}
```

- Qual é a pré-condição pra essa ação?
  - A pré-condição para essa ação é que o jogo ainda não tenha acabado

Ações - MoveO e MoveX

- Por enquanto, as ações `MoveO` e `MoveX` são bem parecidas porque ambos jogam "de qualquer jeito". Não vamos parametrizar elas porque depois vamos mudar somente o comportamento de `X`.

```
action MoveO = all {
  nextTurn == O,
  MoveToEmpty(O),
  nextTurn' = X,
}
```

```
action MoveX = all {
  nextTurn == X,
  MoveToEmpty(X),
  nextTurn' = O,
}
```

- Qual é a pré-condição pra essas ações?

Ações - MoveO e MoveX

- Por enquanto, as ações `MoveO` e `MoveX` são bem parecidas porque ambos jogam "de qualquer jeito". Não vamos parametrizar elas porque depois vamos mudar somente o comportamento de `X`.

```
action MoveO = all {
  nextTurn == O,
  MoveToEmpty(O),
  nextTurn' = X,
}
```

```
action MoveX = all {
  nextTurn == X,
  MoveToEmpty(X),
  nextTurn' = O,
}
```

- Qual é a pré-condição pra essas ações?
  - Para ambas, a pré-condição é que seja o turno do jogador a fazer a jogada
    - Implicitamente, também temos a pré-condição de `MoveToEmpty` empregada nessa ação

Estado inicial

```
action init = all {
  // X always goes first
  nextTurn' = X,
  // Every space in the board starts blank
  board' = 1.to(3).mapBy(_ => 1.to(3).mapBy(_ => Empty
    )),
}
```

Transições

```
action step = any {
  MoveX,
  MoveO,
  // If the game is over, we don't need to do anything
  all { gameOver, board' = board, nextTurn' = nextTurn
     },
}
```

Rodando jogos aleatórios com o simulador

```
quint run tictactoe.qnt --max-samples=1
```

Rodando jogos aleatórios com o simulador

```
quint run tictactoe.qnt --max-samples=1
```

```
...

[State 20]
{
  board:
    Map(
      1 -> Map(1 -> Occupied(O), 2 -> Occupied(X), 3 -> Occupied(X)),
      2 -> Map(1 -> Occupied(X), 2 -> Occupied(O), 3 -> Occupied(O)),
      3 -> Map(1 -> Occupied(X), 2 -> Occupied(O), 3 -> Occupied(X))
    ),
  nextTurn: O
}
```

Usando uma invariante para procurar jogos que “dão velha”

“Dar velha”, ou *stalemate*, quer dizer que o tabuleiro está cheio e ninguém ganhou. É um empate.

```
1 val stalemate = boardFull and not(won(X)) and not(won(
      O))
2
3 val NotStalemate = not(stalemate)
```

Usando uma invariante para procurar jogos que “dão velha”

“Dar velha”, ou *stalemate*, quer dizer que o tabuleiro está cheio e ninguém ganhou. É um empate.

```
val stalemate = boardFull and not(won(X)) and not(won(O))

val NotStalemate = not(stalemate)
```

Essa invariante é fácil de quebrar, podemos usar o simulador ao invés do *model checker* tranquilamente:

```
quint run tictactoe.qnt --invariant=NotStalemate
```

Usando uma invariante para procurar jogos que “dão velha”

“Dar velha”, ou *stalemate*, quer dizer que o tabuleiro está cheio e ninguém ganhou. É um empate.

```
val stalemate = boardFull and not(won(X)) and not(won(O))

val NotStalemate = not(stalemate)
```

Essa invariante é fácil de quebrar, podemos usar o simulador ao invés do *model checker* tranquilamente:

```
quint run tictactoe.qnt --invariant=NotStalemate
```

Mas podemos usar o *model checker* também! Ele vai demorar mais, porque faz BFS e vai levar um tempo para chegar em jogos com 9 jogadas feitas, que são necessárias para um tabuleiro completo.

```
quint verify tictactoe.qnt --invariant=NotStalemate
```

Contraexemplo

```
[State 9]
{
  board:
    Map(
      1 -> Map(1 -> Occupied(O), 2 -> Occupied(X), 3 -> Occupied(O)),
      2 -> Map(1 -> Occupied(O), 2 -> Occupied(X), 3 -> Occupied(X)),
      3 -> Map(1 -> Occupied(X), 2 -> Occupied(O), 3 -> Occupied(X))
    ),
  nextTurn: O
}
```

Outline

Jogando pra ganhar

- Jogo da velha é um jogo bem simples e fácil
- Ainda quando crianças, enjoamos do jogo, porque percebemos que "sempre dá velha"
- Hipótese: Se um jogador seguir uma certa estratégia, ele nunca perde.
  - Consequência: Se os dois jogadores seguirem essa estratégia, nenhum dos dois perde - "sempre dá velha"

Jogando pra ganhar

- Jogo da velha é um jogo bem simples e fácil
- Ainda quando crianças, enjoamos do jogo, porque percebemos que "sempre dá velha"
- Hipótese: Se um jogador seguir uma certa estratégia, ele nunca perde.
  - Consequência: Se os dois jogadores seguirem essa estratégia, nenhum dos dois perde - "sempre dá velha"

Estratégia:

- A primeira jogada é sempre nos cantos
- As outras jogadas fazem a primeira jogada possível nessa lista de prioridade:
  - Ganhar
  - Bloquear
  - Jogar no centro
  - Preparar uma vitória (preenchendo 2 de 3 quadrados numa fila/coluna/diagonal)
  - Jogada qualquer

Jogando pra ganhar

- Jogo da velha é um jogo bem simples e fácil
- Ainda quando crianças, enjoamos do jogo, porque percebemos que "sempre dá velha"
- Hipótese: Se um jogador seguir uma certa estratégia, ele nunca perde.
  - Consequência: Se os dois jogadores seguirem essa estratégia, nenhum dos dois perde - "sempre dá velha"

Estratégia:

- A primeira jogada é sempre nos cantos
- As outras jogadas fazem a primeira jogada possível nessa lista de prioridade:
  - Ganhar
  - Bloquear
  - Jogar no centro
  - Preparar uma vitória (preenchendo 2 de 3 quadrados numa fila/coluna/diagonal)
  - Jogada qualquer

Vamos implementar essa estratégia para o jogador X, enquanto o jogador O continua jogando "de qualquer jeito".

Começando com os cantos

```
1  pure val corners = Set(
2    (1,1),
3    (3,1),
4    (1,3),
5    (3,3)
6  )
7
8  action StartInCorner =
9    nondet corner = oneOf(corners)
10   Move(X, corner)
```

Condições para as jogadas

Precisamos definir as condições que determinam se cada uma das jogadas na lista de prioridade pode ser feita.

- Ganhar
- Bloquear
- Jogar no centro
- Preparar uma vitória

```
val canWin = winningPatterns.exists(canWinWithPattern)
val canBlock = winningPatterns.exists(canBlockWithPattern)
val canTakeCenter = isEmpty((2,2))
val canSetupWin = winningPatterns.exists(canSetupWinWithPattern)
```

(`canWinWithPattern`, `canBlockWithPattern` e `canSetupWintWithPattern` a seguir)

Condições para as jogadas - definições auxiliares

Dado um *winning pattern*, podemos **ganhar** com aquele *pattern* sse duas das coordenadas tiverem `X` e a outra estiver vazia. Lembrando que a ordem não importa.

```
1 def canWinWithPattern(pattern) = and {
2   pattern.filter(coordinate => coordinate.hasPlayer(X)
    ).size() == 2,
3   pattern.filter(coordinate => coordinate.isEmpty()).
    size() == 1,
4 }
```

Condições para as jogadas - definições auxiliares

Dado um *winning pattern*, podemos **ganhar** com aquele *pattern* sse duas das coordenadas tiverem `X` e a outra estiver vazia. Lembrando que a ordem não importa.

```
def canWinWithPattern(pattern) = and {
  pattern.filter(coordinate => coordinate.hasPlayer(X)
    ).size() == 2,
  pattern.filter(coordinate => coordinate.isEmpty()).
    size() == 1,
}
```

Dado um *winning pattern*, podemos **bloquear** com aquele *pattern* sse duas das coordenadas tiverem `O` e a outra estiver vazia.

```
def canBlockWithPattern(pattern) = and {
  pattern.filter(coordinate => coordinate.hasPlayer(O)
    ).size() == 2,
  pattern.filter(coordinate => coordinate.isEmpty()).
    size() == 1,
}
```

Condições para as jogadas - definições auxiliares II

Dado um *winning pattern*, podemos **preparar uma vitória** com aquele *pattern* sse uma das coordenadas tiver `X` e as outras duas estiverem vazias.

```
1 def canSetupWinWithPattern(pattern) = and {
2   pattern.filter(coordinate => coordinate.hasPlayer(X)
    ).size() == 1,
3   pattern.filter(coordinate => coordinate.isEmpty()).
    size() == 2,
4 }
```

Ações - Win

```
action Win = all {
  canWin,
  nondet pattern = winningPatterns.filter(
    canWinWithPattern).oneOf()
  nondet coordinate = pattern.filter(isEmpty).oneOf()
  Move(X, coordinate),
}
```

- Qual é a pré-condição pra essa ação?

Ações - Win

```
action Win = all {
  canWin,
  nondet pattern = winningPatterns.filter(
    canWinWithPattern).oneOf()
  nondet coordinate = pattern.filter(isEmpty).oneOf()
  Move(X, coordinate),
}
```

- Qual é a pré-condição pra essa ação?
  - `canWin`, lembrando que `canWin` é definido por:

```
val canWin = winningPatterns.exists(
  canWinWithPattern)
```

Ações - Win

```
action Win = all {
  canWin,
  nondet pattern = winningPatterns.filter(
    canWinWithPattern).oneOf()
  nondet coordinate = pattern.filter(isEmpty).oneOf()
  Move(X, coordinate),
}
```

- Qual é a pré-condição pra essa ação?
  - `canWin`, lembrando que `canWin` é definido por:

```
val canWin = winningPatterns.exists(
  canWinWithPattern)
```

  - Isso é importante para garantir que nunca estamos chamando `oneOf` em um set vazio.

Ações - Block

```
action Block = all {
  canBlock,
  nondet pattern = winningPatterns.filter(
    canBlockWithPattern).oneOf()
  nondet coordinate = pattern.filter(isEmpty).oneOf()
  Move(X, coordinate),
}
```

Ações - Block

```
action Block = all {
  canBlock,
  nondet pattern = winningPatterns.filter(
    canBlockWithPattern).oneOf()
  nondet coordinate = pattern.filter(isEmpty).oneOf()
  Move(X, coordinate),
}
```

Observem o uso de `oneOf` para selecionar a coordenada aqui. Nesses casos (tanto `Win` quanto `Block`), essa seleção é **determinística**, porque sabemos que sempre haverá uma única coordenada vazia nesses *patterns*. Contudo, o Quint não sabe disso.

Ações - Block

```
action Block = all {
  canBlock,
  nondet pattern = winningPatterns.filter(
    canBlockWithPattern).oneOf()
  nondet coordinate = pattern.filter(isEmpty).oneOf()
  Move(X, coordinate),
}
```

Observem o uso de `oneOf` para selecionar a coordenada aqui. Nesses casos (tanto `Win` quanto `Block`), essa seleção é **determinística**, porque sabemos que sempre haverá uma única coordenada vazia nesses *patterns*. Contudo, o Quint não sabe disso.

- Não existe algo como "pegar o primeiro elemento do set" - porque sets não são ordenados!

Ações - TakeCenter e SetupWin

```
action TakeCenter = Move(X, (2, 2))

action SetupWin = all {
  nondet pattern = winningPatterns.filter(
    canSetupWinWithPattern).oneOf()
  nondet coordinate = pattern.filter(isEmpty).oneOf()
  Move(X, coordinate),
}
```

Ações - alterando MoveX

Temos todas as ações para a estratégia definidas, agora basta definir um novo `MoveX` que chama essas ações conforme a prioridade estabelecida.

Ações - alterando MoveX

Temos todas as ações para a estratégia definidas, agora basta definir um novo `MoveX` que chama essas ações conforme a prioridade estabelecida.

```
action MoveX = all {
  nextTurn == X,
  if (boardEmpty) StartInCorner else
  if (canWin) Win else
  if (canBlock) Block else
  if (canTakeCenter) TakeCenter else
  if (canSetupWin) SetupWin else
  MoveToEmpty(X),
  nextTurn' = O,
}
```

Invariantes

Com isso, temos nosso modelo. Agora, vamos definir algumas invariantes para o uso dessa estratégia.

```
1  /// X has not won. This does not hold, as X wins
     most of the times.
2  val XHasNotWon = not(won(X))
3
4  /// O has not won. This should hold, as O can only
     achieve a draw.
5  val OHasNotWon = not(won(O))
```

Fórmulas temporais

```
1 /// This is not always true, as if O picks the right
      moves, the game will
2 /// result in a stalemate.
3 temporal XMustEventuallyWin = eventually(won(X))
```

- Infelizmente, a implementação de propriedades temporais no Apalache ainda é bem rudimentar.
- Podemos traduzir Quint pra TLA+ e usar o TLC para checar essa propriedade
  - Existe um script para chamar o TLC com os parâmetros apropriados, que podemos usar enquanto a integração não está pronta
    - `sh tlc/check_with_tlc.sh --file tictactoe.qnt --temporal XMustEve`
- O simulador não suporta fórmulas temporais
  - Poderia suportar parcialmente com aquela implementação que fizemos em C++/Haskell na disciplina

Tarefa de casa

Tarefa para a próxima aula: ler o blogpost
`https://elliotswart.github.io/pragmaticformalmodeling/`

- Serve como uma revisão de alguns conteúdos da matéria até agora
- Explica a modelagem do jogo da velha em TLA+, que veremos na próxima aula
- Também conta como referência pra essa aula :)

FIM

Jogo da Velha em Quint

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC

16 de abril de 2025