Lean: Across the Board

Release 0.1.0

Julian Berman

Sep 28, 2020

CONTENTS

A fun attempt at learning some more Lean by implementing some theorems from the (great) Across the Board: The Mathematics of Chessboard Problems book.

CHAPTER

ONE

CONTENTS

1.1 API Reference

1.1.1 `chess.board`

Definitions and theorems about a chess board

Summary

The chess board is a set of indexed *piece`s on a `playfield*. A board is valid, and can only be constructed, if all the pieces are present on the board, and no two distinct (by index) pieces share the same position on the board.

Main definitions

1. The *board* itself, which requires an indexed vector of *piece`s, and the `playfield* which will serve as the where those pieces are placed. Additionally, all pieces must be present on the playfield, and no two distinct (by index) pieces can share a position on the playfield.

2. A way to reduce the board, following the indices to just the pieces. This allows comparison of boards that are equivalent modulo permutation of indices that point to equivalent pieces.

Implementation notes

1. A *board* requires finite dimensions for the *playfield*, finite indices, and a finite piece set. Ideally, this should be generizable to potentially infinite types. However, since *playfield`s are usually provided by `matrix*, which is restricted to finite dimensions, it is easiest to define the board as finite. Additionally, to perform position math, more constraints on the dimension types will likely be necessary, like *decidable_linear_order*.

2. The requirement of *decidable_eq* on the dimensions and index allows use of *dec_trivial* to automatically infer proofs for board constraint propositions. That means instantiation of a *board* will not require explicit proofs for the propositions.

3. The board does not define what are valid position comparisons – the geometry of the space is not defined other than what the *playfield* provides.

4. Currently, all pieces are constrained by the definition of a board to be present on the playfield. That means no capturing moves and no piece introduction moves are possible.

`chess.board`
: A board is axiomatized as a set of indexable (ergo distinguishable) pieces which are placed on distinct squares of a *playfield*.

`chess.board.board_repr`
: A board's representation is just the concatentation of the representations of the *pieces* and *contents* via *board_repr_pieces* and *board_repr_contents*, respectively, with newlines inserted for clarity.

chess.board.board_repr_contents
A board's *contents* can be represented by reducing the board according to the indexed vector at *pieces*, and placing the pieces on the *playfield*. We override the default *option K* representation by using *option_wrap*, and supply an underscore to represent empty positions.

chess.board.board_repr_instance

chess.board.board_repr_pieces
A board's *pieces* is a "vector", so *vec_repr* is used to represent it.

chess.board.has_equiv

chess.board.has_mem

chess.board.height
The height of the board.

chess.board.reduce
The state of the board, where pieces of the same type are equivalent

chess.board.width
The width of the board.

chess.option_wrap
Construct an *option_wrapper* term from a provided *option K* and the *string* that will override the *has_repr.repr* for *none*.

chess.option_wrapper
An auxiliary wrapper for *option K* that allows for overriding the *has_repr* instance for *option*, and rather, output just the value in the *some* and a custom provided *string* for *none*.

chess.wrapped_option_repr

1.1.2 chess.move

chess.move
A move is a (distinct) start and end square whose start square is occupied and whose end square is not.

(Captures are not implemented yet.)

chess.move.after_occupied_end
End squares are occupied after a move.

chess.move.after_unoccupied_start
Start squares are unoccupied after a move.

chess.move.before_after_same
Other squares are unchanged after a move.

chess.move.before_occupied_start
Start squares are occupied before a move.

chess.move.before_unoccupied_end
End squares are unoccupied before a move.

chess.move.no_superimpose
Pieces do not become superimposed after a move.

chess.move.perform_move
A valid *move* on a *board* retains a valid board state.

chess.move.piece
The piece that is being moved.

chess.move.retains_pieces
Pieces do not disappear after a move.

chess.move.start_square_is_some

chess.split_eq

1.1.3 chess.piece

Chess piece implementation.

chess.black_bishop

chess.black_king

chess.black_knight

chess.black_pawn

chess.black_queen

chess.black_rook

chess.color

chess.color.decidable_eq

chess.colored_pieces

chess.colored_pieces.decidable_eq

chess.has_repr

chess.piece_repr

chess.pieces

chess.pieces.decidable_eq

chess.white_bishop

chess.white_king

chess.white_knight

chess.white_pawn

chess.white_queen

chess.white_rook

1.1.4 chess.playfield

Definitions and theorems about the chess board field

Summary

The field on which chess pieces are placed is a 2D plane, where each position corresponds to a piece index. This is because we think of defining pieces and moves, usually, by indicating which position they are at, and which position they are moved to.

Main definitions

1. The playfield itself (*playfield*)
2. Conversion from a *matrix* of (possibly) occupied spaces to a *playfield*

3. Moving a piece by switching the indices at two specified positions using *move_piece*

Implementation details

1. The *playfield* type itself has no requirements to be finite in any dimension, or that the indices used are finite. We represent the actual index wrapped by *option*, such that the empty square can be an *option.none*. The playfield definition wraps the two types used to define the dimensions of the board into a pair.

2. In the current implementation, the way to construct a *playfield* is to provide a matrix. This limits the *playfield* to a finite 2D plane. Another possible implementation is of a "sparse matrix", where for each index, we can look up where the piece is. This now allows for an infinite playfield, but still complicates using infinite pieces. For now, the closely-tied *matrix* definition makes *playfield* a light type wrapper on top of *matrix*, i.e. a function of two variables.

3. Currently, *move_piece* just swaps the (potentially absent) indices at two positions. This is done by using an *equiv.swap* as an updating function. For now, this means that moves that use *move_piece* are non-capturing. Additionally, no math or other requirements on the positions or their contents is required. This means that *move_piece* supports a move from a position to itself. A separate *move* is defined in *chess.move* that has more chess-like rule constraints.

4. Index presence on the board is not limited to have each index on at-most-one position. Preventing duplication of indices is not enforced by the *playfield* itself. However, any given position can hold at-most-one index on it. The actual chess-like rule constraints are in *chess.board*.

matrix_to_playfield
A conversion function to turn a bare *matrix* into a *playfield*. A *matrix* requires the dimensions to be finite.

An example empty 3 × 3 playfield for 4 pieces could be generated by:

```
matrix_to_playfield ((
  ![![none, none, none],
    ![none, none, none],
    ![none, none, none]] : matrix (fin 3) (fin 3) (option (fin 4))
```

where the positions are 0-indexed, with the origin in the top-left, first dimension for the row, and second dimension for the column (0,0) (0,1) (0,2) (1,0) (1,1) (1,2) (2,0) (2,1) (2,2)

playfield
A *playfield m n* represents a *matrix (m × n) option* , which is a model for a *m* × *n* shaped game board where not every square is occupied.

playfield.has_mem
A piece, identified by an index, is on the board, if there is any position such that the index at that position is the one we're inquiring about. Providing a *has_mem* instance allows using *ix pf* for *ix : , pf : playfield m n* . This definition does not preclude duplicated indices on the playfield. See "Implementation details".

playfield.inhabited
A *playfield* is by default *inhabited* by empty squares everywhere.

playfield.matrix_repr
For a *matrix ^(m' × n')* where the has a *has_repr* instance itself, we can provide a *has_repr* for the matrix, using *vec_repr* for each of the rows of the matrix. This definition is used for displaying the playfield, when it is defined via a *matrix*, likely through notation.

TODO: redefine using a fold + intercalate

playfield.matrix_repr_instance

playfield.move_piece
Move an (optional) index from *start_square* to *end_square* on a *playfield*, swapping the indices at those squares.

Does not assume anything about occupancy.

playfield.move_piece_def
Equivalent to to *move_piece*, but useful for *rewrite*ing.

playfield.move_piece_diff
Moving an (optional) index retains whatever (optional) indices were at other squares.

playfield.move_piece_end
Moving an (optional) index that was at *end_square* places it at *start_square*

playfield.move_piece_start
Moving an (optional) index that was at *start_square* places it at *end_square*

playfield.playfield_repr_instance

playfield.vec_repr
For a "vector" ^*n'* represented by the type *n' : , fin n'* → , where the has a *has_repr* instance itself, we can provide a *has_repr* for the "vector". This definition is used for displaying rows of the playfield, when it is defined via a *matrix*, likely through notation.

TODO: redefine using a fold + intercalate

playfield.vec_repr_instance

1.1.5 guarini

"Proof" of Guarini's Problem: swapping some knights.

Given a board like:

```
_ _ _ _ _
```

Guarini's problem asks for a sequence of moves that swaps the knights, producing:

```
_ _ _ _ _
```

Solution:

```
 _       _       _ _       _       _ _
_ _ _  →   _ _  →   _ _  →  _ _ _  →  _ _
 _       _ _       _       _       _


       _       _ _       _ _       _ _
      →  _ _   →  _ _   →   _   →   _ _
         _  _      _       _  _       _


       _               _       _  _
      →   _ _  →   _ _  →   _   →   _
         _  _     _ _ _     _ _ _     _  _


         _                       _
      →   _   →  _ _   →  _ _ _  →  _ _ _
         _ _ _     _ _ _      _ _      _
```

ending_position

first_move

guarini_position

guarini_seq

guarini_seq.scan_contents

starting_position

vector.scanl

vector.scanr

INDEX