Lean: Across the Board

Release 0.1.0

Julian Berman

Jul 25, 2023

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 `pieces` 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 `pieces`, 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.
3. `board.piece_at`, which extracts the piece which sits on a given square.

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 `playfields` are usually provided by `matrix`, which is restricted to finite dimensions, it is easiest to define the board as finite.
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.

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

No inhabited instance because the index type can be larger than the cardinality of the playfield dimensions.

Fields:

field pieces

field contents

field contains

field injects

def 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.

def 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.

def board_repr_instance
A board's representation is provided by `board_repr`.

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

def contents_decidable
Explicitly state that the proposition that an index `ix :` is in the board is `decidable`, when the is itself `decidable_eq`.

def has_equiv

def has_mem

def height
The height of the board. Explicit argument for projection notation.

theorem inj_iff
Given that the board is `occupied_at` some `pos : m × n`, then the index at some `pos' : m × n` is equal to the index at `pos`, iff that `pos'` is equal `pos' = pos`.

theorem no_superimposed
A board maps each index `ix :` to a unique position `pos : m × n`, stated explicitly. Uses the `board.injects` constraint.

def piece_at
The (colored) `piece` on a given square.

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

theorem retains_pieces
A board contains all of the `ix :` indices that it knows of, stated explicitly. Uses the `board.contains` constraint.

def width
The width of the board. Explicit argument for projection notation.

1.1.2 chess.move

Definitions and theorems about chess board movements

Summary

A move on a particular board is a pair of squares whose start square contains a piece and whose end square does not.

Moves may be combined into sequences of moves, which encapsulate multiple sequential moves all iteratively satisfying the above condition.

Main definitions

1. The move itself, which requires specifying the particular board it will occur on
2. perform_move, which yields the board whose playfield has the start and end squares of a move suitably modified
3. A move sequence, rooted on a starting board, containing a sequence of start and end squares which can be treated as iterated moves.

Implementation notes

1. move and sequence are implemented independently of each other. sequence.moves can be used to extract a move from a particular index into a sequence. sequences are also currently finite, and therefore also may automatically infer proofs of move conditions via dec_trivial.
2. Currently, no legality checks or piece math whatsoever is performed, meaning moves are not yet programmatically confirmed to be legal. Captures are similarly not yet supported.

def chess.board.has_sequence_len
Assert the existence of a sequence of length o from a start_board to a given end board.

def chess.board.has_sequence_to
Assert the existence of a sequence from a start_board to a given end board.

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

No inhabited instance because the board might be made up of a single occupied position.

(Captures are not implemented yet.)

Fields:

field start_square

field end_square

field occupied_start

field unoccupied_end

theorem after_occupied_end
End squares are occupied after a move.

theorem after_unoccupied_start
Start squares are unoccupied after a move.

theorem before_after_same
Other squares are unchanged after a move.

theorem before_after_same_occupied
: Other occupation are unchanged after a move.

theorem before_occupied_start
: Start squares are occupied before a move.

theorem before_unoccupied_end
: End squares are unoccupied before a move.

def decidable_eq

theorem diff_squares
: The start and end squares of a move are distinct.

def fintype

theorem no_superimposed
: Pieces do not become superimposed after a move.

def perform_move
: A valid `move` on a `board` retains a valid board state.

def piece
: The piece that is being moved.

theorem retains_injectivity
: A `move` retains accesing indices injectively on the `board` it operates on.

theorem retains_surjectivity
: A `move` retains all indices, ignoring empty squares, present on the `board` it operates on.

def scan_contents
: Define the mapping of `playfield`s after performing successive `move_piece`s using the pairs of positions in the provided `elements`, starting from the `start_board`.

constant sequence
: A move `sequence` represents a sequential set of moves from a starting `board`.

  No inhabited instance because boards do not have an inhabited instance.

  Fields:

  field start_board

  field elements

  field all_occupied_start'

  field all_unoccupied_end'

theorem sequence.all_occupied_start
: Every scanned board is occupied at the start square of the upcoming move.

theorem sequence.all_unoccupied_end
: Every scanned board is unoccupied at the end square of the upcoming move.

def sequence.boards
: The board which results from applying the first $\texttt{ix}_0 + 1$ `move`s in the `sequence`.

def sequence.contents_at
: Shorthand for referring to the contents at a sequence index `ix : fin (o + 1)`.

theorem sequence.contents_at_def
: Shorthand for referring to the contents at a sequence index `ix : fin (o + 1)`.

def sequence.end_board
The board which results from applying all moves in the sequence.

theorem sequence.fixes_unmentioned_squares
Any square which is not the start_square or end_square of any move in the sequence is fixed across all moves (i.e. contains the same piece or remains empty).

def sequence.moves
The ix_0'th move in the sequence.

theorem sequence.no_superimposed
Pieces do not become superimposed after any move in a sequence.

theorem sequence.retains_injectivity
Every playfield in a sequence of moves injectively accesses the indices.

theorem sequence.retains_pieces
Pieces do not disappear after any move_piece in a sequence.

theorem sequence.retains_surjectivity
Every playfield in a sequence of moves contains all the indices it can.

theorem sequence.sequence_step
Any contents_at a step in the sequence is the result of performing a move_piece using the sequence.elements at that step.

theorem sequence.sequence_zero
The first contents in a scan_contents sequence is of the start_board.

1.1.3 chess.move.legal

Legal chess move definitions and theorems

Summary

Legal chess moves are moves which satisfy the legal rules of chess. This includes both the mathematics of which squares a given piece type can move to and the broader set of board conditions that must be satisfied (e.g. not moving a king into check).

Only a subset of these rules are currently implemented below so far. Currently:

- knight move math

are what is implemented.

(No chess variants are currently implemented either.)

Main definitions

1. move.is_legal, which can decide whether a given move is legal
2. move.legal, which represents a move along with the above proof that the move.is_legal
3. board.moves_from, which given a position on the provided board, produces the set of legal moves which may be performed from that square.

Implementation notes

1. moves_from is currently defined to return a finset, even though in theory topologically one could have boards with infinitely many immediate next squares. This finiteness assumption will eventually need fixing in other places, so it seems safe here for now.
2. The requirement of decidable_eq on the various types surrounding move.legal means that again dec_trivial can automatically infer proofs for move legality without them being explicitly provided.

theorem chess.board.mem_moves_from
The finset of legal moves from a given square.

def chess.board.moves_from
The finset of legal moves from a given square.

def chess.board.moves_from.fintype

theorem chess.board.moves_from_def
The finset of legal moves from a given square.

def chess.move.adjacent
Two squares pos and pos' are adjacent (i.e. have no square between them).

def chess.move.adjacent.decidable_pred

def chess.move.between
The finite set of (presumably squares) between two elements of m (or n).

def chess.move.is_legal
A legal chess move.

def chess.move.is_legal_decidable

def chess.move.knight_move
A legal knight move moves 2 squares in one direction and 1 in the other.

def chess.move.knight_move.decidable_pred

constant chess.move.legal
A legal move is a move along with a proof that the move satisfies the rules of chess.

Fields:

field to_move

field legality

def fintype

def chess.move.one_gap
Two squares pos and pos' have exactly one square between them.

def chess.move.one_gap.decidable_pred

constant chess.move.sequence.legal
Fields:

field to_sequence

field legality

theorem chess.moves_from.unoccupied_zero
There are 0 legal moves from an unoccupied square.

1.1.4 chess.piece

Chess piece implementation.

def chess.black_bishop

def chess.black_king

def chess.black_knight

def chess.black_pawn

def chess.black_queen

def chess.black_rook

constant chess.color

def chess.color.decidable_eq

def chess.color.fintype

constant chess.colored_piece
Fields:

field piece

field color

def chess.colored_piece.decidable_eq

def chess.colored_piece.fintype

def chess.has_coe
"Forget" a piece's color.

def chess.has_repr

constant chess.piece

def decidable_eq

def fintype

def chess.piece_repr

def chess.white_bishop

def chess.white_king

def chess.white_knight

def chess.white_pawn

def chess.white_queen

def chess.white_rook

1.1.5 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`
4. Making a sequence of moves at once using `move_sequence`

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`.
5. Sequences of moves are implemented on top of `move`s, rather than vice versa (`move`s being defined as sequences of length one). This *probably* causes a bit of duplication, which may warrant flipping things later.

`def 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)

def 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.

theorem playfield.coe_occ_val
A `pos : pf.occupied_positions` can be used as a `pos : m × n`.

def playfield.decidable_pred
The predicate that `pf.occupied_at pos` for some pos is decidable if the indices `ix :` are finite and decidably equal.

theorem playfield.exists_of_occupied
A `pos : pf.occupied_positions' has the property that there is an not-necessarily-uniqueix : such thatpf pos = some ix`.

theorem playfield.exists_unique_of_occupied
A `pos : pf.occupied_positions' has the property that there is a necessarily-uniqueix : such thatpf pos = some ix`.

theorem playfield.finite_occupied
When the `playfield` dimensions are all finite, the `occupied_positions_set` of all positions that are `occupied_at` is a `fintype`.

def playfield.fintype

def playfield.fintype_occupied
When the `playfield` dimensions are all finite, the `occupied_positions_set` of all positions that are `occupied_at` is finite.

def playfield.has_coe

def 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 : ℕ, pf : playfield m n`. This definition does not preclude duplicated indices on the playfield. See "Implementation details".

def playfield.index_at
Extract the `ix :` that is at `pf pos = some ix`.

theorem playfield.index_at.implies_surjective
Index retrieval via `pf` is known to be surjective, given an surjectivity condition via `function.surjective pf.index_at` and an unoccupied square somewhere.

theorem playfield.index_at.injective
Index retrieval via `pf.index_at` is known to be injective, given an injectivity condition via `pf.some_injective`.

theorem playfield.index_at.surjective
Index retrieval via `pf.index_at` is known to be surjective, given an surjectivity condition via `function.surjective pf`.

theorem playfield.index_at_def
Extract the `ix :` that is at `pf pos = some ix`.

theorem playfield.index_at_exists
The index retrieved via `pf.index_at` is known to be in the `pf`, in existential format.

theorem playfield.index_at_exists'
The index retrieved via `pf.index_at` is known to be in the `pf`, in existential format, operating on the `pf.occupied_positions` subtype.

theorem playfield.index_at_iff
For a `pos : pf.occupied_positions`, the wrapped index `ix :` given by `pf.index_at pos` is precisely `pf pos`, in iff form.

theorem playfield.index_at_in
The index retrieved via `pf.index_at` is known to be in the `pf`.

theorem playfield.index_at_inj
Index retrieval via `pf.index_at` is known to be injective, given an injectivity condition via `pf.some_injective`.

theorem playfield.index_at_inv_pos_from'
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, the right inverse of `pf.index_at` is `pf.pos_from'`.

theorem playfield.index_at_mk
For a `pos : m × n`, and the hypothesis that `h : pf pos = some ix`, the index given by `pf.index_at (occupied_positions.mk _ h)` is precisely `ix`.

theorem playfield.index_at_retains_surjectivity
If every index and the empty square is present in the `pf : playfield m n`, as given by a `function.surjective pf` proposition, then each index is present on the playfield after a `move_piece`.

theorem playfield.index_at_some
For a `pos : pf.occupied_positions`, the wrapped index given by `pf.index_at pos` is precisely `pf pos`.

def playfield.index_equiv
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, there is an explicit equivalence from the indices to the type of positions in `pf.occupied_positions`.

def playfield.inhabited
A `playfield` is by default `inhabited` by empty squares everywhere.

theorem playfield.inj_iff
When a `pf : playfield m n` is `some_injective`, if it is occupied at some `pos : m × n`, then it is injective at that `pos`.

theorem playfield.inj_on_occupied
The injectivity of `some_injective` is equivalent to the `set.inj_on` proposition.

theorem playfield.injective
When a `pf : playfield m n` is `some_injective`, if it is not empty at some `pos : m × n`, then it is injective at that `pos`.

def 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.

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

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

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

theorem playfield.move_piece_occupied_diff
The `pf : playfield m n` is `occupied_at other_square` after a `move_piece`, for a `pos` that is neither `start_square` nor `end_square`, iff it is `occupied_at other_square` before the piece move.

theorem playfield.move_piece_occupied_end
The `pf : playfield m n` is `occupied_at end_square` after a `move_piece` iff it is `occupied_at start_square` before the piece move.

theorem playfield.move_piece_occupied_start
The `pf : playfield m n` is `occupied_at start_square` after a `move_piece` iff it is `occupied_at end_square` before the piece move.

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

def playfield.move_sequence
Make a sequence of `move`s all at once.

theorem playfield.move_sequence_def
Equivalent to to `move_sequence`, but useful for `rewrite` ing.

theorem playfield.move_sequence_diff
Throughout a sequence, moving an (optional) index retains whatever (optional) indices that were at other squares on the next board.

theorem playfield.move_sequence_end
Throughout a sequence, moving an (optional) index that was at `end_square` places it at `start_square` on the next board.

theorem playfield.move_sequence_start
Throughout a sequence, moving an (optional) index that was at `start_square` places it at `end_square` on the next board.

theorem playfield.nonempty_pos
Given a surjectivity condition of `pf.index_at`, the type of `pos : pf.occupied_positions` that identify a particular index is a nonempty.

theorem playfield.not_occupied_at_iff
A `pos : m × n` is unoccupied iff it is `none`.

theorem playfield.not_occupied_has_none
If for some `pf : playfield m n`, at `pos : m × n`, `pf pos = none`, then that is equivalent to `¬ pf.occupied_at pos`.

def playfield.occ_set_decidable
The predicate that `p, p pf.occupied_position_set` for some pos is decidable if the indices `ix :` are finite and decidably equal.

def playfield.occupied_at
A wrapper to indicate that there is some `ix :` such that for a `pf : playfield m n`, at `pos : m × n`, `pf pos = some ix`.

theorem playfield.occupied_at_def
A wrapper to indicate that there is some `ix :` such that for a `pf : playfield m n`, at `pos : m × n`, `pf pos = some ix`.

theorem playfield.occupied_at_iff
A wrapper to indicate that there is some `ix :` such that for a `pf : playfield m n`, at `pos : m × n`, `pf pos = some ix`.

theorem playfield.occupied_at_of_ne
If for some `pf : playfield m n`, at `pos : m × n`, `pf pos none`, then that is equivalent to `pf.occupied_at pos`.

theorem playfield.occupied_at_of_some
If for some `pf : playfield m n`, at `pos : m × n`, `pf pos = some ix`, then that is equivalent to `pf.occupied_at pos`.

theorem playfield.occupied_at_transfer
If for some `pf : playfield m n`, at `pos : m × n`, `pf.occupied_at pos`, then for a `pos' : m × n` such that `pf pos = pf pos'`, we have that `pf.occupied_at pos'`.

theorem playfield.occupied_at_unique
A `pf : playfield m n` maps any occupied `pos` uniquely.

def playfield.occupied_fintype
The `occupied_positions` of a `pf : playfield m n` are finite if the dimensions of the playfield and the indices are finite.

theorem playfield.occupied_has_not_none
A wrapper API for converting between inequalities and existentials.

theorem playfield.occupied_has_some
A wrapper API for underlying `option.is_some` propositions.

theorem playfield.occupied_is_some
A `pos : pf.occupied_positions'` has the property that `pf pos` is occupied.

def playfield.occupied_position_finset
The `finset` of all positions that are `occupied_at`, when all the dimensions of the `playfield` are `fintype`.

def playfield.occupied_positions
The `set` of all positions that are `occupied_at`.

def playfield.occupied_positions.mk
Given some `ix :` such that for `pf : playfield m n` and `pos : m × n`, `pf pos = some ix`, we can subtype into `pos : pf.occupied_positions`.

theorem playfield.occupied_positions_def
Given some `ix :` such that for `pf : playfield m n` and `pos : m × n`, `pf pos = some ix`, we can subtype into `pos : pf.occupied_positions`.

theorem playfield.occupied_positions_in
The `pos : m × n` that is in `pf.occupied_positions` by definition is the proposition that `pf.occupied_at pos`.

theorem playfield.occupied_some_injective
The injectivity of `pf.some_injective` extends to the `pf.occupied_positions` subtype.

theorem playfield.occupied_unique_of_injective
The index retrieved via `pf.index_at` is known to be unique in the `pf`, given an injectivity condition via `pf.some_injective`.

def playfield.playfield_decidable_in

def playfield.playfield_repr_instance

def playfield.pos_from
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, the type there exists a `pos : m × n' such that`pf pos = some ix`.

def playfield.pos_from'
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, we can retrieve the `pos : pf.occupied_positions` such that `pf.index_at pos = ix`.

theorem playfield.pos_from.injective
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, the function `pf.pos_from` is injective.

theorem playfield.pos_from_at
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, roundtripping to get the `pf (pf.pos_from ix _ _)` is exactly `some ix`,

theorem playfield.pos_from_at'
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, roundtripping to get the `pf (pf.pos_from' ix _ _)` is exactly `some ix`, which goes through the coercion down to `pos : m × n`.

def playfield.pos_from_aux
A helper subtype definition describing all the positions that match an index.

No inhabited instance exists because the type could be empty, if none of the positions of the playfield have this index.

theorem playfield.pos_from_aux_subtype
A helper subtype definition describing all the positions that match an index.

def playfield.pos_from_auxf
A helper finset definition describing all the positions that match an index.

theorem playfield.pos_from_auxf_finset
A helper finset definition describing all the positions that match an index.

theorem playfield.pos_from_auxf_in
A helper finset definition describing all the positions that match an index.

theorem playfield.pos_from_auxf_set
A helper set definition describing all the positions that match an index.

theorem playfield.pos_from_def
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, the type there exists a `pos : m × n' such that`pf pos = some ix`.

theorem playfield.pos_from_def'
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, we can retrieve the `pos : pf.occupied_positions` such that `pf.index_at pos = ix`.

theorem playfield.pos_from_index_at'
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, roundtripping to get the `pf.index_at (pf.pos_from' ix _ _)` is exactly `ix`.

theorem playfield.pos_from_inv
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, the partial inverse of `pf.pos_from` is `pf` itself.

theorem playfield.pos_from_inv_index_at'
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, the left inverse of `pf.index_at` is `pf.pos_from'`.

theorem playfield.pos_from_occupied
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, the position retrieved via `pf.pos_from` means that the `pf` is `occupied_at` it.

theorem playfield.retains_injectivity
Each index that is present on the playfield and appears only once, appears only once after a `move_piece`.

theorem playfield.retains_pieces
Pieces do not disappear after a `move_piece`.

theorem playfield.retains_surjectivity
If every index and the empty square is present in the `pf : playfield m n`, as given by a `function.surjective pf` proposition, then each index is present on the playfield after a `move_piece`.

def playfield.some_injective
A `playfield` on which every index that appears, appears only once.

def playfield.some_injective_decidable
Explicitly state that the proposition that `pf.some_injective` is `decidable`, when the is itself `decidable_eq`.

theorem playfield.subsingleton_pos
Given an injectivity condition of `pf.some_injective`, the type of `pos : pf.occupied_positions` that identify a particular index is a subsingleton.

theorem playfield.unique_of_injective
When a `pf : playfield m n` is `some_injective`, every index `ix :` `pf` exists in the `pf` uniquely.

theorem playfield.unique_of_occupied
When a `pf : playfield m n` is `some_injective`, every `pos : pf.occupied_positions` maps to a unique index via `pf pos`.

theorem playfield.unique_pos
Given a surjectivity condition of `pf.index_at`, and an injectivity condition of `pf.some_injective`, the type of `pos : pf.occupied_positions` that identify a particular index is a unique.

1.1.6 chess.utils

Helpers that don't currently fit elsewhere…

def matrix_repr
For a `matrix X^(m' × n')` where the `X` 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.

def matrix_repr_instance

def option_wrap
Construct an `option_wrapper` term from a provided `option X` and the `string` that will override the `has_repr.repr` for `none`.

constant option_wrapper
An auxiliary wrapper for `option X` 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`.

Fields:

field val

field none_s

def vec_repr
For a "vector" `X^n'` represented by the type `n' : , fin n' → X`, where the `X` 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.

def vec_repr_instance

def wrapped_option_repr

theorem split_eq

1.1.7 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:

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


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


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


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

def ending_position

def first_move

theorem guarini

def guarini_seq

def starting_position