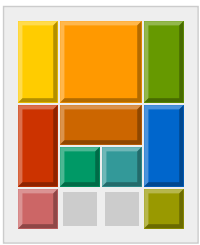
A SOLVER FOR KLOTSKI (210/210 points)

Klotski is a sliding block puzzle, that has a fairly detailed page on WikiPedia. The purpose of this project is to write a solver for Klotski using a graph exploration algorithm.

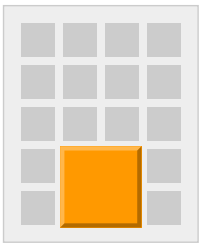
The Klotski puzzle is made of a board of 4x5 places which contains the following 10 pieces:

- one 2x2 square piece (written `S`);
- one 2x1 horizontal rectangle piece (written `H`);
- four 1x2 vertical rectangle pieces (written `V0`, `V1`, `V2` and `V3`);
- four 1x1 square pieces (written `C0`, `C1`, `C2`, `C3`).

The puzzle is presented in the initial configuration that is shown in the following picture:

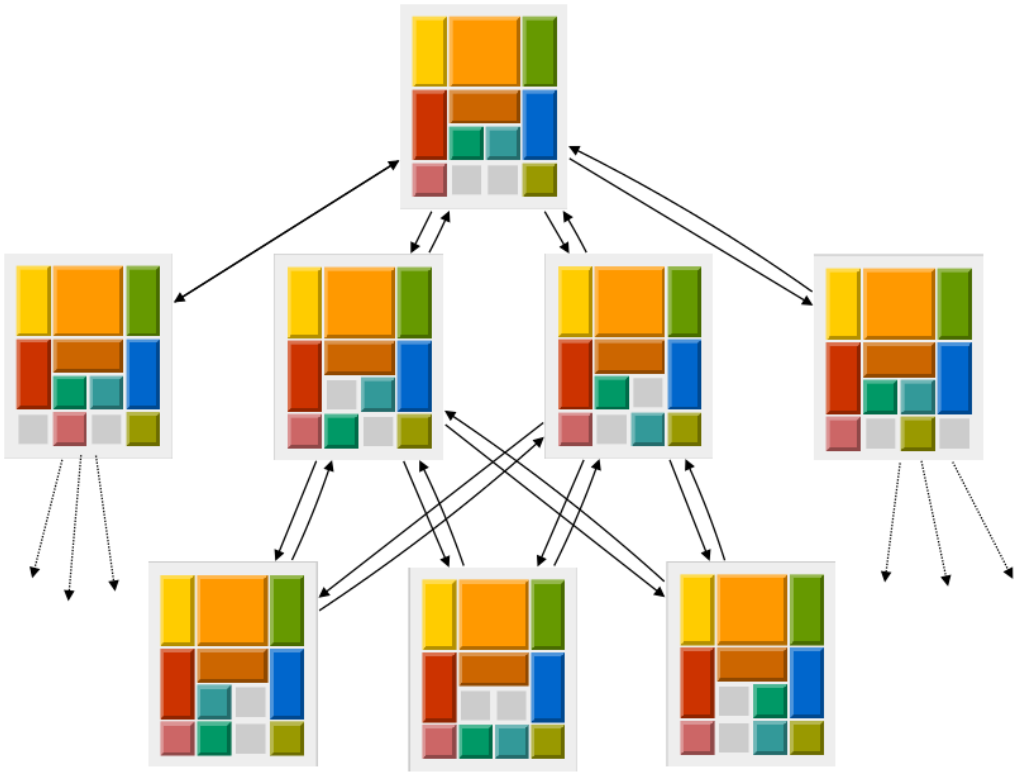


A move in this puzzle consists in sliding a piece at a time, and to win you must have managed to move the large square piece all the way down, reaching a configuration that matches the following shape pattern:



To solve the puzzle, one may search all its state space for a winning configuration. The state space of the Klotski puzzle can be described as a graph, having as nodes the configurations of the board, and as arrows the possible moves.

In the following picture, we show an excerpt of the Klotski graph, starting from the initial configuration. All the boards reachable from the initial configuration in one step are shown, as well as a few of the boards reachable in two steps.



Since all moves can be undone, it is easy to see that one can move from any configuration of the graph to any other configuration of the graph. This means that, when exploring the graph, one will naturally pass through the same configuration more than once, following different paths. To avoid spending our time running in circles when searching for a solution, we will need to avoid visiting a configuration more than once!

APPROACH

To solve the Klotski puzzle, you will proceed in two steps:

A. As a first step, you will write a *generic solver* for any problem whose search space can be represented by a graph, and making sure you handle properly loops in the graph.

B. As a second step, you will describe the Klotski puzzle search space as a graph.

Once the two steps above are done, finding a solution will just be a matter of passing the Klotski graph to the generic solver.

> **Note:** this project may take quite a lot of time to be graded, because it is long, and because the algorithm is complex. We suggest that you use the typecheck button and the toplevel extensively, so that you are reasonnably sure of your code before submitting it to the tests. Also, we added a function `grade_only : int list -> unit`, that you may call in your code to select the exercises to grade. All other exercises won't be graded at all, and considered failed. For instance, if you write `grade_only [ 3 ] ;;` at the beginning of your file, only exercise 3 will be tested.

PRELIMINARIES

First, you will implement some useful basic functions.

1. Write a function `loop` of type `('a -> bool) -> ('a -> 'a) -> 'a -> 'a` such that `loop p f x = x` when `p x = true` and `loop p f x = loop p f (f x)` otherwise.
2. Write a function `exists` of type `('a -> bool) -> 'a list -> bool` such that `exists p l = true` if and only if there exists an element `x` of `l` such that `p x = true`.
3. Write a function `find` of type `('a -> bool) -> 'a list -> 'a` such that `find p l = x` if `x` is the first element of `l` for which `p x = true`. If no such

element exists, `find` raises the exception `NotFound` given in the prelude.

PART A: A GENERIC PROBLEM SOLVER

The goal of this section is to implement a generic problem solver based on graph exploration. The questions will help you start with a naive implementation and refine it step-by-step.

A general way to look at problem solving is to consider some set E representing the states of the problem and to consider also a finite binary relation $\mathscr{R}$ that represents the small reasoning steps that can be made to move from one state of the problem to another.

Remember that a binary relation is a subset of all the pairs in E. We will write $x\mathscr{R}y$ if (x, y) is in $\mathscr{R}$. The image of x under $\mathscr{R}$ in E is written $\mathscr{R}(x)$ and it is defined as the set of all y in E such that $x\mathscr{R}y$. A relation can be viewed through this image function as a function from E to the subsets of E. Hence, we can use the following type definition (also given in the prelude) to represent binary relations.

```
type 'e rel = 'e -> 'e list
```

4. As an exercise, we want to define a relation $x\mathscr{N}y$ that tells if the difference between two integers is at most 2. For instance, we have that $1\mathscr{N}3$, $3\mathscr{N}1$ and $2\mathscr{N}2$, but not $1\mathscr{N}4$ or $5\mathscr{N}0$.
   Define `near: int rel` that encodes the image of this relation as an OCaml function. For instance, `near 2` should return something like `[0;1;2;3;4]`.

In practice, the `'e rel` type describes the function which, from a given configuration of the problem, gives all the possible configurations in which we can end up after performing a step. In this question, we want not just to look at the possible next steps from a single configuration, but from a set of configurations.
Formally, we will say that we extend the image function $\mathscr{R}$ of a binary relation $\mathscr{R}$ over E to a function $\overline{\mathscr{R}}$ defined as follows.
$\overline{\mathscr{R}}([]) = []$ and $\overline{\mathscr{R}}(x::xs) = \mathscr{R}(x)@\overline{\mathscr{R}}(xs)$.
Basically, this computes the list of all possible new configuration that are reachable in one step from any configuration in an original list, losing track of the exact predecessor of each configuration. We just know that if a configuration is present in the resulting set, then there must have been one in the original set that led to it in one step.

5. Write function `flat_map` of type `'e rel -> ('e list -> 'e list)` such that `flat_map r` represents $\overline{\mathscr{R}}$ if `r` represents a binary relation $\mathscr{R}$.
   For instance, `flat_map near` applied to `[2;3;4]` should return something like `[0;1;2;3;4;1;2;3;4;5;2;3;4;5;6]`.

A binary relation over the set of problem configurations relates all pairs of configurations that are separated by a single step (for us, moving one piece of the game).
Sometimes, we want to relate a configuration with its possible futures, up to a given number of steps.
Formally, if $\mathscr{R}$ if a binary relation over E, we say that $x\mathscr{R}^n y$ iff there exists a chain of elements e_i of E of length $n-1$ such as $x\mathscr{R}e_1\mathscr{R}...\mathscr{R}e_{n-1}\mathscr{R}y$. The image function of $\mathscr{R}^n$ is simply the image function of $\mathscr{R}$ iterated n times.

6. Write a function `iter_rel: 'e rel -> int -> 'e rel` that computes this iteration.
   Iterating a relation 1 time or less does nothing (identity).
   For instance, `iter_rel near 2` should be the image function of the relation that tells is two integers are separated by 4 of less.

The transitive closure of a binary relation $\mathscr{R}$ is the relation that iterates $\mathscr{R}$ over and over.
Therefore, this is the union of all the relations obtained by iterating n times the relation $\mathscr{R}$, for all n.
Formally, $\mathscr{R}^*(x) = \mathscr{R}^0(x) \cup \mathscr{R}^1(x) \cup \mathscr{R}^2(x) \cup ...$
Or more constructively, $\mathscr{R}^0(x) = [x]$ and $\mathscr{R}^{n+1}(x) = \overline{\mathscr{R}}(\mathscr{R}^n(x))$.

We are not interested in computing the transitive closure of any relation, which could not terminate, depending on the relation. Our purpose is to compute the possible futures, starting from the initial configuration of the problem, until a specific property of the state is reached (in

our case, we won the game).

To represent such a property, we use the following type (also given in the prelude). It is a function that takes a state, and tells if the property holds (returning `true`) or not.

```
type 'e prop = 'e -> bool
```

Solving a problem characterized by a property p and a relation $\mathscr{R}$ for an initial problem state s is finding an element x in $\mathscr{R}(s)$ such that $p(x) =$ `true`.

7. Write a function `solve` of type `'a rel -> 'a prop -> 'a -> 'a` such that `solve r p x` computes the iteration of the relation $\mathscr{R}$ represented by `r` starting at `x` until it reaches an element `y` such that `p y`.
   For instance, `solve near (fun x -> x = 12) 0` should simply return `12`. Internally, the function will start from the set `[0]`, and iterate `near` to obtain first `[-2;-1;0;1;2]`, then a sequence of growing lists, until eventually one iteration returns a list that contains `12`.
8. Define a function `solve_path` of type `'a rel -> 'a prop -> 'a -> 'a list` such that `solve_path r p x` behaves exactly as `solve` except that it produces not only the final value `y` such that `p y` but also all the intermediate elements from `x` to `y` that show how `x` is related to `y` through `r`.
   For instance, `solve_path near (fun x -> x = 12) 0` should return `[0;2;4;6;8;10;12]`.
   This function can be written simply by calling `solve` with well-chosen arguments. The idea is to iterate over the set of paths to destinations, instead of the set of destinations.

The previous solver is very naive since it introduces a lot of redundancy in the search process. This is due to the simplistic representation of sets as lists: an element may be repeated several times in a list. Let us assume that we are given a more efficient data structure for sets over elements of type `'a` as specified by the following record type `('a, 'set) set_operations` (also given in the prelude).

```
type ('a, 'set) set_operations =
  { empty : 'set ;
    mem : 'a -> 'set -> bool ;
    add : 'a -> 'set -> 'set }
```

This is a pattern that you will see in some OCaml libraries, to combine a set of operations in something more syntactically lightweight than an object or a functor. It is simple to use: given that you have a value `ops` of this type, just use `ops.empty` to obtain an empty set, `ops.add 13 s` to create a new set that is a copy of an existing set `s` augmented with a new element `13`, `s.add 1 (s.add 2 s.empty)` to create a set from scratch containing two elements `1` and `2`, and finally `s.mem 8 s` to test if `8` is an element of a set `s`.

From now on, we will assume that a value of type `('a, 'set) set_operations` will be provided as input to the solver (you won't have to write one). For your own tests, you can use `int_set_operations: (int, _) set_operations` and `int_list_set_operations: (int list, _) set_operations` that are predefined.

9. Write a function `archive_map` of type `('a, 'set) set_operations -> 'a rel -> ('set * 'a list) -> ('set * 'a list)` such that: `archive_map opset rel (s, l) = (s', l')`, where:
   - `l'` is the list of elements that are reachable using `rel` from the elements of `l` and which are not already in the set `s`.
   - `s'` is the union of `s` and the elements of `l'`.
10. Use `archive_map` to program a new function `solve'` (don't forget the quote after the name) of type `('a, 'set) set_operations -> 'a rel -> 'a prop -> 'a -> 'a` that explores the search space with no redundancy.

11. Same question for `solve_path'` of type `('a list, 'set) set_operations -> 'a rel -> 'a prop -> 'a -> 'a list`.

The last step of this first part is to implement a solver for a one-player puzzle. A one-player puzzle is characterized by the following elements.

i. a type for `'configuration`s which represent the states of the puzzle ;

ii. a type for `'move`s which represent how the player can move from one configuration to another ;

iii. a function `move : 'configuration -> 'move -> 'configuration` which applies a move to a configuration to get a new one ;

iv. a function `possible_moves : 'configuration -> 'move list` which returns the list of moves that can be applied to an input configuration ;

v. a function `final : 'configuration -> bool` which returns `true` if and only if the input configuration is the one we are looking for.

The following record type `puzzle` (also given in the prelude) will be used to characterize a puzzle.

```
type ('configuration, 'move) puzzle =
  { move : 'configuration -> 'move -> 'configuration;
    possible_moves : 'configuration -> 'move list;
    final : 'configuration -> bool }
```

12. Implement `solve_puzzle : ('c, 'm) puzzle -> ('c list, 's) set_operations -> 'c -> 'c list` such that `solve_puzzle p opset c` is a list of moves the application of which to the configuration `c` results in a configuration `c'` such that `p.final c' = true`. Here, `opset` is a record of set operations which the implementation of `solve_puzzle` may use.

PART B: A SOLVER FOR KLOTSKI

Let's start by giving names to the pieces on the board:

- the one 2x2 square piece are written `S` ;
- the one 2x1 horizontal rectangle piece are written `H` ;
- the four 1x2 vertical rectangle pieces are written `V0`, `V1`, `V2` and `V3` ;
- the four 1x1 square pieces are written `C0`, `C1`, `C2`, `C3`.

As you have surely noticed, the board has two places which are not covered by any piece. We will write `X` to denote such a place.

The type to describe the different kind of pieces is naturally an enumeration:

```
type piece_kind = S | H | V | C | X
```

A piece is characterized by its kind and an index.

```
type piece = piece_kind * int
```

Let us enumerate the pieces using toplevel definitions:

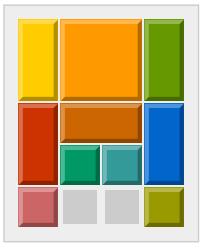
```
let x = (X, 0) and s = (S, 0) and h = (H, 0)
let (c0, c1, c2, c3) = ((C, 0), (C, 1), (C, 2), (C, 3))
let (v0, v1, v2, v3) = ((V, 0), (V, 1), (V, 2), (V, 3))
let all_pieces : piece list = [ s; h; c0; c1; c2; c3; v0; v1; v2; v3 ]
```

A board is a matrix of 5x4 places. Each place refers to the piece that covers it.

```
type board = piece array array
```

You can use the function `display_board: board -> unit` to visualize a board configuration in the toplevel.

For instance, the initial configuration is written as follows:



```
[| [| (V,0) ; (S,0) ; (S,0) ; (V,1) |] ;
   [| (V,0) ; (S,0) ; (S,0) ; (V,1) |] ;
   [| (V,2) ; (H,0) ; (H,0) ; (V,3) |] ;
   [| (V,2) ; (C,0) ; (C,1) ; (V,3) |] ;
   [| (C,2) ; (X,0) ; (X,0) ; (C,3) |] |]
```

13. Write a function `final: board -> bool` such that `final board = true` if and only if `board` is a final configuration for Klotski.

We have defined the configurations of our puzzle, now we have to define the moves. We will use the following types (also given in the prelude).

```
type move = Move of piece * direction * board
and direction = { dcol : int; drow : int }
```

A move is characterized by a direction and a piece it is applied to. The third component is the board that is obtained when this move is applied to the current board.

With this definition of a move, applying a move to a board is very simple: we just extract the image board from it.

```
let move _ (Move (_, _, b)) = b
```

14. Write a function `move_piece : board -> piece -> direction -> board option` such that `move_piece board p { drow; dcol } = Some board'` if moving the piece `p` in `board` in direction `{ drow; dcol }` is possible and gives `board'`. Otherwise, if this move is not possible, `move_piece` returns `None`.
15. Define `possible_moves : board -> move list` that returns a list of moves that can be applied to `board`.

At this point, you can define the puzzle instance to pass to the generic solver.

```
let klotski : (board, move) puzzle = { move; possible_moves; final }
```

16. The solver also expects a data structure to represent sets of boards. Use the standard library's functor `Set.Make` to implement this data structure as a module `BoardSet` of signature `Set.S with type elt = board`.
    The required comparison function `compare : board -> board -> int` should be such that `compare b1 b2 = 0` if all cells of both arrays are exactly the same.
    Otherwise, it should compare the cells of the two arrays pairwise, starting from cell `.(0).(0)`, then `.(0).(1)`, `.(0).(2)`, `.(0).(3)`, `.(1).(0)`, etc. up to the last cell `.(4).(3)`. The function should return the result of the first comparison of cells that does not return `0`.
    To compare to cells at index `.(i).(j)`, first the first component (the kind of piece) is compared. The result is `< 0` if `b1.(i).(j) < b2.(i).(j)` and `> 0` if `b1.(i).(j) > b2.(i).(j)`. For this the pieces are ordered as follows: `S > H > C > V > X`.
    If both first components are the same, the result is the comparison of the second component, with the usual order on integers.
17. Update your `compare` function, so that it performs as few array accesses as possible, respecting the previous ordering algorithm. It must read the arrays up to the first pair of cells that differ. It should only read the entire arrays in case of equality.

Hint: If you used loops, you can use an exception.
Note: All array accesses are counted, so `a.(y).(x)` counts for two. If you read several cells in a same row, better put the row in a variable!

PUTTING IT ALL TOGETHER

18. Write a function `solve_klotski : board -> board list` such that `solve_klotski initial_board` is a list of boards from the `initial_board` to a `board` such that `final board`. This list must come from a sequence of valid moves of this puzzle.

You can use the function `display_solution: board list -> unit` to visualize a solution in the toplevel. The two values `initial_board_trivial` and `initial_board_simpler` are variants of the initial configuration whose resolution should be much faster.

THE GIVEN PRELUDE

```
exception NotFound

type 'e rel = 'e -> 'e list
type 'e prop = 'e -> bool

type ('a, 'set) set_operations = {
  empty : 'set;              (* The empty set. *)
  mem : 'a -> 'set -> bool;  (* [mem x s = true] iff [x] is in [s]. *)
  add : 'a -> 'set -> 'set;  (* [add s x] is the set [s] union {x}. *)
}

type ('configuration, 'move) puzzle = {
  move : 'configuration -> 'move -> 'configuration;
  possible_moves : 'configuration -> 'move list;
  final : 'configuration -> bool
}

type piece_kind = S | H | V | C | X
type piece = piece_kind * int
let x = (X, 0) and s = (S, 0) and h = (H, 0)
let (c0, c1, c2, c3) = ((C, 0), (C, 1), (C, 2), (C, 3))
let (v0, v1, v2, v3) = ((V, 0), (V, 1), (V, 2), (V, 3))
```



```
let initial_board =
  [| [| v0 ; s  ; s  ; v1 |];
     [| v0 ; s  ; s  ; v1 |];
     [| v2 ; h  ; h  ; v3 |];
     [| v2 ; c0 ; c1 ; v3 |];
     [| c2 ; x  ; x  ; c3 |] |]

let initial_board_simpler =
  [| [| c2 ; s  ; s  ; c1 |] ;
     [| c0 ; s  ; s  ; c3 |] ;
     [| v1 ; v2 ; v3 ; v0 |] ;
     [| v1 ; v2 ; v3 ; v0 |] ;
     [| x  ; x  ; x  ; x  |] |]

let initial_board_trivial =
  [| [| x  ; s  ; s  ; x  |] ;
     [| x  ; s  ; s  ; x  |] ;
     [| x  ; x  ; x  ; x  |] ;
     [| x  ; x  ; x  ; x  |] ;
     [| x  ; x  ; x  ; x  |] |]

type direction = { dcol : int; drow : int; }
type move = Move of piece * direction * board
let move _ (Move (_, _, b)) = b
```

YOUR OCAML ENVIRONMENT

```
let rec loop p f x =
  if p x = true then  x  else loop p f (f x)
;;

let rec exists p l = match l with
  | [] -> false
  | hd::tl -> if (p hd) then true else exists p tl
;;

let rec find p l = match l with
  | [] -> raise NotFound
  | hd::tl -> if (p hd) then hd else find p tl
;;

(* --- Part A: A Generic Problem Solver --- *)

let near x =
  let rec construct nb ref l =
    if abs((nb) - ref) <= 2 then construct (nb + 1) ref ((nb)::l) else l
  in
  construct (x - 2) x []
;;

let rec flat_map r = function
  | [] -> []
  | x::xs -> ((r x)@(flat_map r xs))
;;

let rec iter_rel rel n =  function
  | x ->  if n <= 1 then rel x else flat_map rel ((iter_rel rel (n - 1)) x)
;;
```

Evaluate >

Switch >>

Typecheck

Reset Templ

Full-screen

Check & Sa

Exercise complete (click for details) — 210 pts

- ∨ Preliminaries — Completed, 30 pts
  - ∨ Exercise 1: `loop` — Completed, 10 pts
    - Found `loop` with compatible type.
    - Computing `loop odd succ 95`
    - Correct value 95 — 1 pt
    - Computing `loop even pred 117`
    - Correct value 116 — 1 pt
    - Computing `loop (power_of 3) pred 110`
    - Correct value 81 — 1 pt
    - Computing `loop (power_of 2) succ 176`
    - Correct value 256 — 1 pt
    - Computing `loop even succ 46`
    - Correct value 46 — 1 pt
    - Computing `loop (power_of 2) pred 105`
    - Correct value 64 — 1 pt

    - Correct value 143 — 1 pt
    - Computing `loop (power_of 3) pred 117`
    - Correct value 81 — 1 pt
    - Computing `loop even succ 127`
    - Correct value 128 — 1 pt
  - ∨ Exercise 2: `exists` — Completed, 10 pts
    - Found `exists` with compatible type.
    - Computing `exists (power_of 2) []`
    - Correct value `false` — 1 pt
    - Computing `exists odd [11]`
    - Correct value `true` — 1 pt
    - Computing `exists (power_of 2) [14; 10]`
    - Correct value `false` — 1 pt
    - Computing `exists odd [1; 10; 7]`
    - Correct value `true` — 1 pt
    - Computing `exists even [3; 8; 6]`
    - Correct value `true` — 1 pt
    - Computing `exists (power_of 3) [16; 6; 9; 4]`
    - Correct value `true` — 1 pt
    - Computing `exists odd [7; 15; 2; 6]`
    - Correct value `true` — 1 pt
    - Computing `exists even [1; 11]`
    - Correct value `false` — 1 pt
    - Computing `exists (power_of 2) [1; 15]`

Correct value `true` 1 pt
Computing `exists (power_of 3) [14; 8; 15; 1]`
Correct value `true` 1 pt

v Exercise 3: `find` Completed, 10 pts

Found `find` with compatible type.
Computing `find (power_of 2) []`
Correct exception `NotFound` 1 pt
Computing `find even [1; 15; 16]`
Correct value `16` 1 pt
Computing `find odd [7; 11; 5; 1]`
Correct value `7` 1 pt
Computing `find (power_of 3) [5; 11]`
Correct exception `NotFound` 1 pt
Computing `find odd [16; 6]`
Correct exception `NotFound` 1 pt
Computing `find (power_of 3) [16; 8; 10]`
Correct exception `NotFound` 1 pt
Computing `find even [9; 7]`
Correct exception `NotFound` 1 pt
Computing `find (power_of 2) [3; 11; 5]`
Correct exception `NotFound` 1 pt
Computing `find odd [3; 5; 13]`
Correct value `3` 1 pt
Computing `find (power_of 2) [4]`
Correct value `4` 1 pt

v Part A: A Generic Problem Solver Completed, 90 pts

v Exercise 4: `near` Completed, 10 pts

Found `near` with compatible type.
Computing `near 14`
Correct value `[16; 15; 14; 13; 12]` 1 pt
Computing `near 13`
Correct value `[15; 14; 13; 12; 11]` 1 pt
Computing `near 10`
Correct value `[12; 11; 10; 9; 8]` 1 pt
Computing `near 16`
Correct value `[18; 17; 16; 15; 14]` 1 pt
Computing `near 11`
Correct value `[13; 12; 11; 10; 9]` 1 pt

Computing `near 2`
Correct value `[4; 3; 2; 1; 0]` 1 pt
Computing `near 2`
Correct value `[4; 3; 2; 1; 0]` 1 pt
Computing `near 14`
Correct value `[16; 15; 14; 13; 12]` 1 pt
Computing `near 8`
Correct value `[10; 9; 8; 7; 6]` 1 pt

v Exercise 5: `flat_map` Completed, 10 pts

Found `flat_map` with compatible type.
Computing `flat_map id [12]`
Correct value `[12]` 1 pt
Computing `flat_map empty [1; 16; 7]`
Correct value `[]` 1 pt
Computing `flat_map near [11; 13; 7; 5]`
Correct value `[9; 10; 11; 12; 13; 11; 12; 13; 14; 15; 5; 6; 7; 8; 9; 3; 4; 5; 6; 7]` 1 pt
Computing `flat_map id [6]`
Correct value `[6]` 1 pt
Computing `flat_map empty [15; 3; 8]`
Correct value `[]` 1 pt
Computing `flat_map near [14; 6]`
Correct value `[12; 13; 14; 15; 16; 4; 5; 6; 7; 8]` 1 pt
Computing `flat_map near [6; 5; 8; 2]`
Correct value `[4; 5; 6; 7; 8; 3; 4; 5; 6; 7; 6; 7; 8; 9; 10; 0; 1; 2; 3; 4]` 1 pt
Computing `flat_map empty [13; 3; 11]`

Correct value [] 1 pt

Computing flat_map id [1; 4; 9; 3]

Correct value [1; 4; 9; 3] 1 pt

Computing flat_map empty [12; 14; 10; 8]

Correct value [] 1 pt

v Exercise 6: iter_rel Completed, 10 pts

Found iter_rel with compatible type.

Computing iter_rel near 2 2

Correct value 1 pt

```
[-2; -1; 0; 1; 2; -1; 0; 1; 2; 3; 0; 1; 2; 3; 4; 1; 2; 3; 4; 5; 2; 3; 4; 5;
 6]
```

Computing iter_rel id 3 6

Correct value [6] 1 pt

Computing iter_rel near 2 7

Correct value 1 pt

```
[3; 4; 5; 6; 7; 4; 5; 6; 7; 8; 5; 6; 7; 8; 9; 6; 7; 8; 9; 10; 7; 8; 9; 10;
 11]
```

Computing iter_rel near 3 1

Correct value 1 pt

```
[-5; -4; -3; -2; -1; -4; -3; -2; -1; 0; -3; -2; -1; 0; 1; -2; -1; 0; 1; 2;
 -1; 0; 1; 2; 3; -4; -3; -2; -1; 0; -3; -2; -1; 0; 1; -2; -1; 0; 1; 2; -1; 0;
 1; 2; 3; 0; 1; 2; 3; 4; -3; -2; -1; 0; 1; -2; -1; 0; 1; 2; -1; 0; 1; 2; 3;
 0; 1; 2; 3; 4; 1; 2; 3; 4; 5; -2; -1; 0; 1; 2; -1; 0; 1; 2; 3; 0; 1; 2; 3;
 4; 1; 2; 3; 4; 5; 2; 3; 4; 5; 6; -1; 0; 1; 2; 3; 0; 1; 2; 3; 4; 1; 2; 3; 4;
 5; 2; 3; 4; 5; 6; 3; 4; 5; 6; 7]
```

Computing iter_rel id 3 9

Correct value [9] 1 pt

Computing iter_rel empty 3 12

Correct value [] 1 pt

Computing iter_rel near 2 7

Correct value 1 pt

```
[3; 4; 5; 6; 7; 4; 5; 6; 7; 8; 5; 6; 7; 8; 9; 6; 7; 8; 9; 10; 7; 8; 9; 10;
 11]
```

Computing iter_rel empty 3 14

Correct value [] 1 pt

Computing iter_rel id 2 5

Correct value [5] 1 pt

Computing iter_rel empty 2 9

Correct value [] 1 pt

v Exercise 7: solve Completed, 10 pts

Found solve with compatible type.

Computing solve pred (power_of 2) 88

Computing solve succ odd 90

Correct value 91 1 pt

Computing solve pred even 54

Correct value 54 1 pt

Computing solve pred even 50

Correct value 50 1 pt

Computing solve succ (power_of 3) 77

Correct value 81 1 pt

Computing solve pred (power_of 3) 99

Correct value 81 1 pt

Computing solve succ odd 88

Correct value 89 1 pt

Computing solve succ (power_of 3) 63

Correct value 81 1 pt

Computing solve pred (power_of 3) 59

Correct value 27 1 pt

v Exercise 8: solve_path Completed, 10 pts

Found solve_path with compatible type.

Computing solve_path pred odd 75

Correct value [75] 1 pt

Computing solve_path succ even 88

Correct value [88] 1 pt

Computing solve_path pred even 73

Correct value [73; 72] 1 pt

Computing `solve_path succ (power_of 2) 86`

Correct value 1 pt

```
[86; 87; 88; 89; 90; 91; 92; 93; 94; 95; 96; 97; 98; 99; 100; 101; 102; 103;
 104; 105; 106; 107; 108; 109; 110; 111; 112; 113; 114; 115; 116; 117; 118;
 119; 120; 121; 122; 123; 124; 125; 126; 127; 128]
```

Computing `solve_path pred odd 72`

Correct value `[72; 71]` 1 pt

Computing `solve_path succ (power_of 2) 81`

Correct value 1 pt

```
[81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93; 94; 95; 96; 97; 98; 99;
 100; 101; 102; 103; 104; 105; 106; 107; 108; 109; 110; 111; 112; 113; 114;
 115; 116; 117; 118; 119; 120; 121; 122; 123; 124; 125; 126; 127; 128]
```

Computing `solve_path pred (power_of 2) 65`

Correct value `[65; 64]` 1 pt

Computing `solve_path succ odd 85`

Correct value `[85]` 1 pt

Computing `solve_path succ even 66`

Correct value `[66]` 1 pt

Computing `solve_path pred (power_of 3) 56`

Correct value 1 pt

```
[56; 55; 54; 53; 52; 51; 50; 49; 48; 47; 46; 45; 44; 43; 42; 41; 40; 39; 38;
 37; 36; 35; 34; 33; 32; 31; 30; 29; 28; 27]
```

v Exercise 9: `archive_map` Completed, 10 pts

Found `archive_map` with compatible type.

Computing

```
archive_map
  {empty = (int_set_of_list []); mem = mem; add = add}
  id
  ((int_set_of_list [-10;-8;-7;-6;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;8;9]),
   [-2; 3; -10; 4; -8; -5; -6; -4; 6; 1; 2])
```

Correct value `((int_set_of_list [-10;-8;-7;-6;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;8;9]), [])` 1 pt

Computing

```
archive_map
  {empty = (int_set_of_list []); mem = mem; add = add}
  pred
  ((int_set_of_list [-10;-9;-7;-5;-4;-3;-2;-1;0;1;2;3;4;6;8;9]),
   [8; 2; -3; 6; 9; -10; 4; 0; -4])
```

Correct value 1 pt

```
((int_set_of_list [-11;-10;-9;-7;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;7;8;9]),
 [-11; 5; 7])
```

Computing

```
archive_map
  {empty = (int_set_of_list []); mem = mem; add = add}
  succ
  ((int_set_of_list [-10;-9;-8;-7;-6;-5;-4;-3;-2;0;1;2;3;4;5;6;7;8;9]),
   [5; 7; -2; 8; -9; 9; 0; -4; 6; -3; -10])
```

Correct value 1 pt


```
  {empty = (int_set_of_list []); mem = mem; add = add}
  near
  ((int_set_of_list [-9;-1;1;2;3;4;5;6;7;8]), [3; 6; 5; 4; 1; -1; 8])
```

Correct value 1 pt

`((int_set_of_list [-9;-3;-2;-1;0;1;2;3;4;5;6;7;8;9;10]), [10; 9; -2; -3; 0])`

Computing

```
archive_map
  {empty = (int_set_of_list []); mem = mem; add = add}
  pred
  ((int_set_of_list [-4;-2;1;2;4;7]), [2; 1; 4; -2; 7])
```

Correct value `((int_set_of_list [-4;-3;-2;0;1;2;3;4;6;7]), [6; -3; 3; 0])` 1 pt

Computing

```
archive_map
  {empty = (int_set_of_list []); mem = mem; add = add}
  succ
  ((int_set_of_list [-10;-5;0;1;2;3;4]), [-5; -10; 4; 3])
```

Correct value `((int_set_of_list [-10;-9;-5;-4;0;1;2;3;4;5]), [5; -9; -4])` 1 pt

Computing

```
archive_map
  {empty = (int_set_of_list []); mem = mem; add = add}
  near
  ((int_set_of_list [-10;-9;-7;-6;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;7;8;9]),
   [-7; 6; 0; 7; 4; -2; -1; -9; 3; 5])
```

Correct value 1 pt

```
((int_set_of_list [-11;-10;-9;-8;-7;-6;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;7;8;9]),
 [-11; -8])
```

Computing

```
archive_map
  {empty = (int_set_of_list []); mem = mem; add = add}
  id
  ((int_set_of_list [-10;-6;-4;-2;-1;0;1;4;6;8;9]), [-2; -6; 8; 0; 6; -4])
```

Correct value `((int_set_of_list [-10;-6;-4;-2;-1;0;1;4;6;8;9]), [])` 1 pt

```
Computing
  archive_map
    {empty = (int_set_of_list []); mem = mem; add = add}
    near
    ((int_set_of_list [-10;-9;-7;-6;-5;-4;-3;-2;-1;0;2;3;6;8]),
     [-4; -10; 3; -3; -6; 6; -7; 2])
```
Correct value 1 pt
```
  ((int_set_of_list [-12;-11;-10;-9;-8;-7;-6;-5;-4;-3;-2;-1;0;1;2;3;4;5;6;7;8]),
   [7; 5; 4; 1; -8; -11; -12])
Computing
  archive_map
    {empty = (int_set_of_list []); mem = mem; add = add}
    succ
    ((int_set_of_list [-3;-2;-1;0;3;4;5]), [5; 0])
```
Correct value ((int_set_of_list [-3;-2;-1;0;1;3;4;5;6]), [1; 6]) 1 pt

v Exercise 10: `solve'` Completed, 10 pts

Found `solve'` with compatible type.

```
Computing
  solve'
    {empty = (int_set_of_list []); mem = mem; add = add}
    pred
    (power_of 2)
    75
```
Correct value 64 1 pt

Computing `solve' {empty = (int_set_of_list []); mem = mem; add = add} pred odd 62`

Correct value 61 1 pt

```
Computing
  solve'
    {empty = (int_set_of_list []); mem = mem; add = add}
    succ
    (power_of 3)
    90
```
Correct value 243 1 pt

Computing `solve' {empty = (int_set_of_list []); mem = mem; add = add} succ even 96`

Correct value 96 1 pt

```
Computing
  solve'
    {empty = (int_set_of_list []); mem = mem; add = add}
    succ
    (power_of 3)
    68
```
Correct value 81 1 pt

Computing `solve' {empty = (int_set_of_list []); mem = mem; add = add} pred odd 82`

Correct value 81 1 pt

```
Computing
  solve'
    {empty = (int_set_of_list []); mem = mem; add = add}
    succ
```


Computing `solve' {empty = (int_set_of_list []); mem = mem; add = add} pred odd 87`

Correct value 87 1 pt

Computing `solve' {empty = (int_set_of_list []); mem = mem; add = add} succ even 52`

Correct value 52 1 pt

Computing `solve' {empty = (int_set_of_list []); mem = mem; add = add} pred even 93`

Correct value 92 1 pt

v Exercise 11: `solve_path'` Completed, 10 pts

Found `solve_path'` with compatible type.

```
Computing
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    pred
    (power_of 2)
    84
```
Correct value 1 pt
```
  [84; 83; 82; 81; 80; 79; 78; 77; 76; 75; 74; 73; 72; 71; 70; 69; 68; 67; 66;
   65; 64]
Computing
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    succ
    (power_of 2)
    78
```
Correct value 1 pt
```
  [78; 79; 80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93; 94; 95; 96;
   97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109; 110; 111; 112;
   113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124; 125; 126; 127;
   128]
Computing
```

```
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    pred
    even
    59
```
Correct value `[59; 58]` 1 pt

```
Computing
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    pred
    even
    77
```
Correct value `[77; 76]` 1 pt

```
Computing
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    succ
    (power_of 3)
    95
```
Correct value 1 pt

```
  [95; 96; 97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109; 110;
   111; 112; 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124; 125;
   126; 127; 128; 129; 130; 131; 132; 133; 134; 135; 136; 137; 138; 139; 140;
   141; 142; 143; 144; 145; 146; 147; 148; 149; 150; 151; 152; 153; 154; 155;
   156; 157; 158; 159; 160; 161; 162; 163; 164; 165; 166; 167; 168; 169; 170;
   171; 172; 173; 174; 175; 176; 177; 178; 179; 180; 181; 182; 183; 184; 185;
   186; 187; 188; 189; 190; 191; 192; 193; 194; 195; 196; 197; 198; 199; 200;
   201; 202; 203; 204; 205; 206; 207; 208; 209; 210; 211; 212; 213; 214; 215;
   216; 217; 218; 219; 220; 221; 222; 223; 224; 225; 226; 227; 228; 229; 230;
   231; 232; 233; 234; 235; 236; 237; 238; 239; 240; 241; 242; 243]
```

```
Computing
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    succ
    (power_of 2)
    96
```
Correct value 1 pt

```
  [96; 97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109; 110; 111;
   112; 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124; 125; 126;
   127; 128]
```

```
Computing
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    pred
    (power_of 3)
    84
```
Correct value `[84; 83; 82; 81]` 1 pt

```
Computing
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    pred
```



```
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    succ
    (power_of 2)
    59
```
Correct value `[59; 60; 61; 62; 63; 64]` 1 pt

```
Computing
  solve_path'
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    succ
    odd
    71
```
Correct value `[71]` 1 pt

v Exercise 12: `solve_puzzle` Completed, 10 pts

Now I will test your code on a little game.
It is called "Mister Rabbit's Great Escape" .
Mister rabbit is pursued by a dog, and has to find his way home.
He just has to run straight from point (A) to point (B).
In our case, these will just be integers.
But mister Rabbit is easily scared.
So he walks very cautiously: `(fun x -> pos + 1).`
When he is surprised, he runs backward: `(fun x -> pos - 3).`
And on a rush, he can jump very far: `(fun x -> pos * 2).`
To sum up, your solver will be tested on the following:

```
  type rabbit_move = Forward | Backward | Jump
  let rabbit_moves _ =  [ Forward ; Backward ; Jump ]
  let move_rabbit pos = function
    | Forward -> pos + 1
    | Backward -> pos - 3
    | Jump -> pos * 2
```

Found `solve_puzzle` with compatible type.

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 12)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    15
```
Correct value `[15; 12]` 1 pt

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 20)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    19
```
Correct value `[19; 20]` 1 pt

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 32)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    12
```
Correct value `[12; 9; 18; 15; 16; 32]` 1 pt

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 28)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    14
```
Correct value `[14; 28]` 1 pt

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 32)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    16
```
Correct value `[16; 32]` 1 pt

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 15)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    16
```
Correct value `[16; 13; 14; 15]` 1 pt

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 17)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    15
```
Correct value `[15; 16; 17]` 1 pt

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 23)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    10
```
Correct value `[10; 11; 22; 23]` 1 pt

```
Computing
```


Correct value `[11; 22; 23]` 1 pt

```
Computing
  solve_puzzle
    {move = move_rabbit; possible_moves = rabbit_moves; final = ((=) 29)}
    {empty = (int_list_set_of_list []); mem = mem; add = add}
    15
```
Correct value `[15; 16; 32; 29]` 1 pt

∨ Part B: A Solver for Klotski — Completed, 90 pts

∨ Exercise 13: `final` — Completed, 10 pts

Found `final` with compatible type.

```
Computing
  final
    [|[|(C, 3); (H, 0); (H, 0); (V, 0)|]; [|(X, 0); (V, 3); (C, 1); (V, 0)|];
      [|(C, 2); (V, 3); (X, 0); (V, 2)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(V, 1); (S, 0); (S, 0); (C, 0)|]|]
```
Correct value `true` 1 pt

```
Computing
  final
    [|[|(V, 2); (X, 0); (C, 0); (V, 3)|]; [|(V, 2); (H, 0); (H, 0); (V, 3)|];
      [|(S, 0); (S, 0); (C, 3); (V, 0)|]; [|(S, 0); (S, 0); (V, 1); (V, 0)|];
      [|(C, 1); (X, 0); (V, 1); (C, 2)|]|]
```
Correct value `false` 1 pt

```
Computing
  final
    [|[|(C, 1); (V, 1); (V, 0); (C, 2)|]; [|(X, 0); (V, 1); (V, 0); (C, 3)|];
      [|(S, 0); (S, 0); (V, 2); (V, 3)|]; [|(S, 0); (S, 0); (V, 2); (V, 3)|];
      [|(C, 0); (X, 0); (H, 0); (H, 0)|]|]
```
Correct value `false` 1 pt

```
Computing
```

```
  final
    [|[|(X, 0); (C, 1); (V, 0); (V, 2)|]; [|(H, 0); (H, 0); (V, 0); (V, 2)|];
      [|(V, 3); (C, 2); (C, 0); (X, 0)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(C, 3); (S, 0); (S, 0); (V, 1)|]|]
```
Correct value `true` 1 pt

Computing
```
  final
    [|[|(V, 1); (S, 0); (S, 0); (X, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 0)|];
      [|(V, 3); (C, 1); (C, 2); (V, 0)|]; [|(V, 3); (V, 2); (H, 0); (H, 0)|];
      [|(C, 3); (V, 2); (X, 0); (C, 0)|]|]
```
Correct value `false` 1 pt

Computing
```
  final
    [|[|(H, 0); (H, 0); (X, 0); (C, 2)|]; [|(C, 3); (V, 0); (V, 1); (V, 2)|];
      [|(C, 1); (V, 0); (V, 1); (V, 2)|]; [|(V, 3); (S, 0); (S, 0); (C, 0)|];
      [|(V, 3); (S, 0); (S, 0); (X, 0)|]|]
```
Correct value `true` 1 pt

Computing
```
  final
    [|[|(X, 0); (C, 1); (V, 2); (C, 0)|]; [|(H, 0); (H, 0); (V, 2); (V, 0)|];
      [|(C, 3); (C, 2); (X, 0); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 3)|];
      [|(V, 1); (S, 0); (S, 0); (V, 3)|]|]
```
Correct value `true` 1 pt

Computing
```
  final
    [|[|(S, 0); (S, 0); (V, 2); (C, 0)|]; [|(S, 0); (S, 0); (V, 2); (C, 1)|];
      [|(V, 0); (X, 0); (V, 3); (C, 3)|]; [|(V, 0); (C, 2); (V, 3); (V, 1)|];
      [|(H, 0); (H, 0); (X, 0); (V, 1)|]|]
```
Correct value `false` 1 pt

Computing
```
  final
    [|[|(C, 2); (X, 0); (V, 0); (V, 3)|]; [|(S, 0); (S, 0); (V, 0); (V, 3)|];
      [|(S, 0); (S, 0); (V, 1); (V, 2)|]; [|(C, 1); (X, 0); (V, 1); (V, 2)|];
      [|(C, 0); (H, 0); (H, 0); (C, 3)|]|]
```
Correct value `false` 1 pt

Computing
```
  final
    [|[|(H, 0); (H, 0); (V, 3); (C, 0)|]; [|(V, 0); (V, 2); (V, 3); (V, 1)|];
      [|(V, 0); (V, 2); (C, 2); (V, 1)|]; [|(X, 0); (S, 0); (S, 0); (C, 1)|];
      [|(X, 0); (S, 0); (S, 0); (C, 3)|]|]
```
Correct value `true` 1 pt

v Exercise 14: `move_piece` Completed, 10 pts

Found `move_piece` with compatible type.

Computing
```
  move_piece
    [|[|(H, 0); (H, 0); (V, 1); (C, 3)|]; [|(C, 2); (V, 3); (V, 1); (V, 0)|];
      [|(V, 2); (V, 3); (X, 0); (V, 0)|]; [|(V, 2); (S, 0); (S, 0); (C, 0)|];
      [|(C, 1); (S, 0); (S, 0); (X, 0)|]|]
    (V, 3)
```



```
  move_piece
    [|[|(C, 1); (H, 0); (H, 0); (X, 0)|]; [|(S, 0); (S, 0); (X, 0); (C, 3)|];
      [|(S, 0); (S, 0); (C, 2); (V, 2)|]; [|(V, 1); (V, 3); (V, 0); (V, 2)|];
      [|(V, 1); (V, 3); (V, 0); (C, 0)|]|]
    (C, 3)
    {dcol = 0; drow = -1}
```
Correct value 1 pt
```
  (Some
    [|[|(C, 1); (H, 0); (H, 0); (C, 3)|]; [|(S, 0); (S, 0); (X, 0); (X, 0)|];
      [|(S, 0); (S, 0); (C, 2); (V, 2)|]; [|(V, 1); (V, 3); (V, 0); (V, 2)|];
      [|(V, 1); (V, 3); (V, 0); (C, 0)|]|])
```
Computing
```
  move_piece
    [|[|(V, 1); (V, 2); (C, 1); (C, 0)|]; [|(V, 1); (V, 2); (V, 0); (V, 3)|];
      [|(S, 0); (S, 0); (V, 0); (V, 3)|]; [|(S, 0); (S, 0); (H, 0); (H, 0)|];
      [|(X, 0); (X, 0); (C, 3); (C, 2)|]|]
    (C, 3)
    {dcol = 0; drow = 1}
```
Correct value `None` 1 pt

Computing
```
  move_piece
    [|[|(C, 3); (X, 0); (X, 0); (C, 1)|]; [|(V, 0); (V, 1); (H, 0); (H, 0)|];
      [|(V, 0); (V, 1); (C, 0); (V, 3)|]; [|(V, 2); (S, 0); (S, 0); (V, 3)|];
      [|(V, 2); (S, 0); (S, 0); (C, 2)|]|]
    (V, 1)
    {dcol = 0; drow = -1}
```
Correct value 1 pt
```
  (Some
    [|[|(C, 3); (V, 1); (X, 0); (C, 1)|]; [|(V, 0); (V, 1); (H, 0); (H, 0)|];
      [|(V, 0); (X, 0); (C, 0); (V, 3)|]; [|(V, 2); (S, 0); (S, 0); (V, 3)|];
      [|(V, 2); (S, 0); (S, 0); (C, 2)|]|])
```
Computing

```
  move_piece
    [|[|(V, 2); (X, 0); (C, 0); (C, 1)|]; [|(V, 2); (H, 0); (H, 0); (V, 0)|];
      [|(V, 1); (C, 2); (C, 3); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 3)|];
      [|(X, 0); (S, 0); (S, 0); (V, 3)|]|]
    (C, 0)
    {dcol = -1; drow = 0}
```

Correct value 1 pt

```
  (Some
    [|[|(V, 2); (C, 0); (X, 0); (C, 1)|]; [|(V, 2); (H, 0); (H, 0); (V, 0)|];
      [|(V, 1); (C, 2); (C, 3); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 3)|];
      [|(X, 0); (S, 0); (S, 0); (V, 3)|]|])
```

Computing

```
  move_piece
    [|[|(C, 3); (H, 0); (H, 0); (V, 1)|]; [|(C, 0); (S, 0); (S, 0); (V, 1)|];
      [|(X, 0); (S, 0); (S, 0); (V, 2)|]; [|(C, 2); (V, 0); (V, 3); (V, 2)|];
      [|(X, 0); (V, 0); (V, 3); (C, 1)|]|]
    (S, 0)
    {dcol = -1; drow = 0}
```

Correct value None 1 pt

Computing

```
  move_piece
    [|[|(X, 0); (C, 2); (C, 1); (X, 0)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(V, 3); (S, 0); (S, 0); (V, 1)|]; [|(V, 0); (C, 3); (C, 0); (V, 2)|];
      [|(V, 0); (H, 0); (H, 0); (V, 2)|]|]
    (V, 1)
    {dcol = 0; drow = -1}
```

Correct value 1 pt

```
  (Some
    [|[|(X, 0); (C, 2); (C, 1); (V, 1)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(V, 3); (S, 0); (S, 0); (X, 0)|]; [|(V, 0); (C, 3); (C, 0); (V, 2)|];
      [|(V, 0); (H, 0); (H, 0); (V, 2)|]|])
```

Computing

```
  move_piece
    [|[|(H, 0); (H, 0); (C, 0); (V, 1)|]; [|(C, 3); (X, 0); (V, 2); (V, 1)|];
      [|(X, 0); (C, 1); (V, 2); (V, 3)|]; [|(V, 0); (S, 0); (S, 0); (V, 3)|];
      [|(V, 0); (S, 0); (S, 0); (C, 2)|]|]
    (H, 0)
    {dcol = -1; drow = 0}
```

Correct value None 1 pt

Computing

```
  move_piece
    [|[|(V, 3); (X, 0); (H, 0); (H, 0)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(V, 0); (S, 0); (S, 0); (V, 1)|]; [|(V, 0); (C, 1); (V, 2); (C, 2)|];
      [|(C, 3); (X, 0); (V, 2); (C, 0)|]|]
    (S, 0)
    {dcol = -1; drow = 0}
```

Correct value None 1 pt

Computing

```
  move_piece
    [|[|(H, 0); (H, 0); (X, 0); (C, 2)|]; [|(C, 3); (X, 0); (V, 0); (V, 3)|];
      [|(V, 2); (C, 1); (V, 0); (V, 3)|]; [|(V, 2); (S, 0); (S, 0); (V, 1)|];
```



```
  (Some
    [|[|(H, 0); (H, 0); (X, 0); (C, 2)|]; [|(C, 3); (C, 1); (V, 0); (V, 3)|];
      [|(V, 2); (X, 0); (V, 0); (V, 3)|]; [|(V, 2); (S, 0); (S, 0); (V, 1)|];
      [|(C, 0); (S, 0); (S, 0); (V, 1)|]|])
```

v Exercise 15: `possible_moves` Completed, 10 pts

Found `possible_moves` with compatible type.

Computing

```
  possible_moves
    [|[|(V, 3); (C, 2); (C, 3); (X, 0)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(V, 2); (S, 0); (S, 0); (V, 1)|]; [|(V, 2); (C, 0); (C, 1); (V, 0)|];
      [|(X, 0); (H, 0); (H, 0); (V, 0)|]|]
```

Correct value 1 pt

```
  [Move ((V, 2), {dcol = 0; drow = 1},
    [|[|(V, 3); (C, 2); (C, 3); (X, 0)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(X, 0); (S, 0); (S, 0); (V, 1)|]; [|(V, 2); (C, 0); (C, 1); (V, 0)|];
      [|(V, 2); (H, 0); (H, 0); (V, 0)|]|]);
   Move ((V, 1), {dcol = 0; drow = -1},
    [|[|(V, 3); (C, 2); (C, 3); (V, 1)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(V, 2); (S, 0); (S, 0); (X, 0)|]; [|(V, 2); (C, 0); (C, 1); (V, 0)|];
      [|(X, 0); (H, 0); (H, 0); (V, 0)|]|]);
   Move ((C, 3), {dcol = 1; drow = 0},
    [|[|(V, 3); (C, 2); (X, 0); (C, 3)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(V, 2); (S, 0); (S, 0); (V, 1)|]; [|(V, 2); (C, 0); (C, 1); (V, 0)|];
      [|(X, 0); (H, 0); (H, 0); (V, 0)|]|]);
   Move ((H, 0), {dcol = -1; drow = 0}, [|[|(V, ...); ...|]; ...|]); ...]
```

Computing

```
  possible_moves
    [|[|(X, 0); (C, 1); (V, 3); (C, 3)|]; [|(V, 1); (V, 0); (V, 3); (C, 2)|];
      [|(V, 1); (V, 0); (H, 0); (H, 0)|]; [|(X, 0); (S, 0); (S, 0); (V, 2)|];
      [|(C, 0); (S, 0); (S, 0); (V, 2)|]|]
```

Correct value 1 pt

```
  [Move ((V, 1), {dcol = 0; drow = -1},
    [|[|(V, 1); (C, 1); (V, 3); (C, 3)|]; [|(V, 1); (V, 0); (V, 3); (C, 2)|];
      [|(X, 0); (V, 0); (H, 0); (H, 0)|]; [|(X, 0); (S, 0); (S, 0); (V, 2)|];
      [|(C, 0); (S, 0); (S, 0); (V, 2)|]|]);
   Move ((V, 1), {dcol = 0; drow = 1},
    [|[|(X, 0); (C, 1); (V, 3); (C, 3)|]; [|(X, 0); (V, 0); (V, 3); (C, 2)|];
      [|(V, 1); (V, 0); (H, 0); (H, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(C, 0); (S, 0); (S, 0); (V, 2)|]|]);
   Move ((C, 1), {dcol = -1; drow = 0},
    [|[|(C, 1); (X, 0); (V, 3); (C, 3)|]; [|(V, 1); (V, 0); (V, 3); (C, 2)|];
      [|(V, 1); (V, 0); (H, 0); (H, 0)|]; [|(X, 0); (S, 0); (S, 0); (V, 2)|];
      [|(C, 0); (S, 0); (S, 0); (V, 2)|]|]);
   Move ((C, 0), {dcol = 0; drow = -1}, [|[|(X, ...); ...|]; ...|]); ...]
```

Computing

```
  possible_moves
    [|[|(C, 3); (H, 0); (H, 0); (X, 0)|]; [|(C, 1); (V, 3); (V, 2); (V, 0)|];
      [|(V, 1); (V, 3); (V, 2); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (X, 0)|];
      [|(C, 2); (S, 0); (S, 0); (C, 0)|]|]
```

Correct value 1 pt

```
  [Move ((V, 0), {dcol = 0; drow = -1},
    [|[|(C, 3); (H, 0); (H, 0); (V, 0)|]; [|(C, 1); (V, 3); (V, 2); (V, 0)|];
      [|(V, 1); (V, 3); (V, 2); (X, 0)|]; [|(V, 1); (S, 0); (S, 0); (X, 0)|];
      [|(C, 2); (S, 0); (S, 0); (C, 0)|]|]);
   Move ((V, 0), {dcol = 0; drow = 1},
    [|[|(C, 3); (H, 0); (H, 0); (X, 0)|]; [|(C, 1); (V, 3); (V, 2); (X, 0)|];
      [|(V, 1); (V, 3); (V, 2); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 0)|];
      [|(C, 2); (S, 0); (S, 0); (C, 0)|]|]);
   Move ((C, 0), {dcol = 0; drow = -1},
    [|[|(C, 3); (H, 0); (H, 0); (X, 0)|]; [|(C, 1); (V, 3); (V, 2); (V, 0)|];
      [|(V, 1); (V, 3); (V, 2); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (C, 0)|];
      [|(C, 2); (S, 0); (S, 0); (X, 0)|]|]);
   Move ((H, 0), {dcol = 1; drow = 0}, [|[|(C, ...); ...|]; ...|]); ...]
```

Computing

```
  possible_moves
    [|[|(X, 0); (V, 2); (V, 0); (C, 0)|]; [|(V, 3); (V, 2); (V, 0); (X, 0)|];
      [|(V, 3); (S, 0); (S, 0); (C, 3)|]; [|(C, 2); (S, 0); (S, 0); (V, 1)|];
      [|(H, 0); (H, 0); (C, 1); (V, 1)|]|]
```

Correct value 1 pt

```
  [Move ((V, 3), {dcol = 0; drow = -1},
    [|[|(V, 3); (V, 2); (V, 0); (C, 0)|]; [|(V, 3); (V, 2); (V, 0); (X, 0)|];
      [|(X, 0); (S, 0); (S, 0); (C, 3)|]; [|(C, 2); (S, 0); (S, 0); (V, 1)|];
      [|(H, 0); (H, 0); (C, 1); (V, 1)|]|]);
   Move ((C, 3), {dcol = 0; drow = -1},
    [|[|(X, 0); (V, 2); (V, 0); (C, 0)|]; [|(V, 3); (V, 2); (V, 0); (C, 3)|];
      [|(V, 3); (S, 0); (S, 0); (X, 0)|]; [|(C, 2); (S, 0); (S, 0); (V, 1)|];
      [|(H, 0); (H, 0); (C, 1); (V, 1)|]|]);
   Move ((C, 0), {dcol = 0; drow = 1},
    [|[|(X, 0); (V, 2); (V, 0); (X, 0)|]; [|(V, 3); (V, 2); (V, 0); (C, 0)|];
      [|(V, 3); (S, 0); (S, 0); (C, 3)|]; [|(C, 2); (S, 0); (S, 0); (V, 1)|];
      [|(H, 0); (H, 0); (C, 1); (V, 1)|]|])]
```

Computing

```
  possible_moves
```



```
  [Move ((C, 3), {dcol = -1; drow = 0},
    [|[|(C, 2); (C, 0); (X, 0); (V, 1)|]; [|(V, 0); (S, 0); (S, 0); (V, 1)|];
      [|(V, 0); (S, 0); (S, 0); (C, 1)|]; [|(V, 2); (V, 3); (H, 0); (H, 0)|];
      [|(V, 2); (V, 3); (C, 3); (X, 0)|]|]);
   Move ((C, 0), {dcol = 1; drow = 0},
    [|[|(C, 2); (X, 0); (C, 0); (V, 1)|]; [|(V, 0); (S, 0); (S, 0); (V, 1)|];
      [|(V, 0); (S, 0); (S, 0); (C, 1)|]; [|(V, 2); (V, 3); (H, 0); (H, 0)|];
      [|(V, 2); (V, 3); (X, 0); (C, 3)|]|])]
```

Computing

```
  possible_moves
    [|[|(H, 0); (H, 0); (C, 3); (C, 0)|]; [|(X, 0); (C, 2); (V, 3); (V, 0)|];
      [|(V, 1); (C, 1); (V, 3); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(X, 0); (S, 0); (S, 0); (V, 2)|]|]
```

Correct value 1 pt

```
  [Move ((V, 1), {dcol = 0; drow = -1},
    [|[|(H, 0); (H, 0); (C, 3); (C, 0)|]; [|(V, 1); (C, 2); (V, 3); (V, 0)|];
      [|(V, 1); (C, 1); (V, 3); (V, 0)|]; [|(X, 0); (S, 0); (S, 0); (V, 2)|];
      [|(X, 0); (S, 0); (S, 0); (V, 2)|]|]);
   Move ((V, 1), {dcol = 0; drow = 1},
    [|[|(H, 0); (H, 0); (C, 3); (C, 0)|]; [|(X, 0); (C, 2); (V, 3); (V, 0)|];
      [|(X, 0); (C, 1); (V, 3); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(V, 1); (S, 0); (S, 0); (V, 2)|]|]);
   Move ((C, 2), {dcol = -1; drow = 0},
    [|[|(H, 0); (H, 0); (C, 3); (C, 0)|]; [|(C, 2); (X, 0); (V, 3); (V, 0)|];
      [|(V, 1); (C, 1); (V, 3); (V, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(X, 0); (S, 0); (S, 0); (V, 2)|]|])]
```

Computing

```
  possible_moves
    [|[|(C, 0); (X, 0); (V, 2); (V, 0)|]; [|(H, 0); (H, 0); (V, 2); (V, 0)|];
      [|(V, 3); (C, 2); (X, 0); (C, 1)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(C, 3); (S, 0); (S, 0); (V, 1)|]|]
```

Correct value 1 pt

```
  [Move ((V, 2), {dcol = 0; drow = 1},
    [|[|(C, 0); (X, 0); (X, 0); (V, 0)|]; [|(H, 0); (H, 0); (V, 2); (V, 0)|];
      [|(V, 3); (C, 2); (V, 2); (C, 1)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(C, 3); (S, 0); (S, 0); (V, 1)|]|]);
   Move ((C, 2), {dcol = 1; drow = 0},
    [|[|(C, 0); (X, 0); (V, 2); (V, 0)|]; [|(H, 0); (H, 0); (V, 2); (V, 0)|];
      [|(V, 3); (X, 0); (C, 2); (C, 1)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(C, 3); (S, 0); (S, 0); (V, 1)|]|]);
   Move ((C, 1), {dcol = -1; drow = 0},
    [|[|(C, 0); (X, 0); (V, 2); (V, 0)|]; [|(H, 0); (H, 0); (V, 2); (V, 0)|];
      [|(V, 3); (C, 2); (C, 1); (X, 0)|]; [|(V, 3); (S, 0); (S, 0); (V, 1)|];
      [|(C, 3); (S, 0); (S, 0); (V, 1)|]|]);
   Move ((C, 0), {dcol = 1; drow = 0}, [|[|(X, ...); ...|]; ...|]); ...]
```

Computing

```
  possible_moves
    [|[|(C, 3); (C, 0); (X, 0); (C, 1)|]; [|(V, 0); (S, 0); (S, 0); (V, 1)|];
      [|(V, 0); (S, 0); (S, 0); (V, 1)|]; [|(V, 3); (V, 2); (H, 0); (H, 0)|];
      [|(V, 3); (V, 2); (C, 2); (X, 0)|]|]
```

Correct value 1 pt

```
  [Move ((C, 2), {dcol = 1; drow = 0},
    [|[|(C, 3); (C, 0); (X, 0); (C, 1)|]; [|(V, 0); (S, 0); (S, 0); (V, 1)|];
      [|(V, 0); (S, 0); (S, 0); (V, 1)|]; [|(V, 3); (V, 2); (H, 0); (H, 0)|];
      [|(V, 3); (V, 2); (X, 0); (C, 2)|]|]);
   Move ((C, 1), {dcol = -1; drow = 0},
    [|[|(C, 3); (C, 0); (C, 1); (X, 0)|]; [|(V, 0); (S, 0); (S, 0); (V, 1)|];
      [|(V, 0); (S, 0); (S, 0); (V, 1)|]; [|(V, 3); (V, 2); (H, 0); (H, 0)|];
      [|(V, 3); (V, 2); (C, 2); (X, 0)|]|]);
   Move ((C, 0), {dcol = 1; drow = 0},
    [|[|(C, 3); (X, 0); (C, 0); (C, 1)|]; [|(V, 0); (S, 0); (S, 0); (V, 1)|];
      [|(V, 0); (S, 0); (S, 0); (V, 1)|]; [|(V, 3); (V, 2); (H, 0); (H, 0)|];
      [|(V, 3); (V, 2); (C, 2); (X, 0)|]|])]
```

Computing

```
  possible_moves
    [|[|(X, 0); (V, 1); (C, 3); (V, 0)|]; [|(V, 3); (V, 1); (C, 1); (V, 0)|];
      [|(V, 3); (C, 2); (H, 0); (H, 0)|]; [|(V, 2); (S, 0); (S, 0); (X, 0)|];
      [|(V, 2); (S, 0); (S, 0); (C, 0)|]|]
```

Correct value 1 pt

```
  [Move ((V, 3), {dcol = 0; drow = -1},
    [|[|(V, 3); (V, 1); (C, 3); (V, 0)|]; [|(V, 3); (V, 1); (C, 1); (V, 0)|];
      [|(X, 0); (C, 2); (H, 0); (H, 0)|]; [|(V, 2); (S, 0); (S, 0); (X, 0)|];
      [|(V, 2); (S, 0); (S, 0); (C, 0)|]|]);
   Move ((C, 0), {dcol = 0; drow = -1},
    [|[|(X, 0); (V, 1); (C, 3); (V, 0)|]; [|(V, 3); (V, 1); (C, 1); (V, 0)|];
      [|(V, 3); (C, 2); (H, 0); (H, 0)|]; [|(V, 2); (S, 0); (S, 0); (C, 0)|];
      [|(V, 2); (S, 0); (S, 0); (X, 0)|]|])]
```

Computing

```
  possible_moves
    [|[|(V, 3); (V, 1); (V, 2); (C, 3)|]; [|(V, 3); (V, 1); (V, 2); (C, 0)|];
      [|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (V, 0); (C, 2)|];
      [|(X, 0); (C, 1); (V, 0); (X, 0)|]|]
```

Correct value 1 pt

```
  [Move ((C, 2), {dcol = 0; drow = 1},
```



```
    [|[|(V, 3); (V, 1); (V, 2); (C, 3)|]; [|(V, 3); (V, 1); (V, 2); (C, 0)|];
      [|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (V, 0); (C, 2)|];
      [|(C, 1); (X, 0); (V, 0); (X, 0)|]|])]
```

v Exercise 16: `BoardSet` Completed, 20 pts

Found `BoardSet` with compatible type.

Your module has the expected signature. Well done! 5 pts

Now, I will check your comparison.

Found `BoardSet.compare_elt` with compatible type.

Computing

```
  BoardSet.compare_elt
    [|[|(C, 3); (C, 2); (C, 1); (X, 0)|]; [|(V, 2); (S, 0); (S, 0); (V, 0)|];
      [|(V, 2); (S, 0); (S, 0); (V, 0)|]; [|(V, 3); (C, 0); (X, 0); (V, 1)|];
      [|(V, 3); (H, 0); (H, 0); (V, 1)|]|]
    [|[|(V, 1); (V, 0); (V, 2); (V, 3)|]; [|(V, 1); (V, 0); (V, 2); (V, 3)|];
      [|(X, 0); (X, 0); (H, 0); (H, 0)|]; [|(C, 0); (S, 0); (S, 0); (C, 1)|];
      [|(C, 2); (S, 0); (S, 0); (C, 3)|]|]
```

Correct value 1 1 pt

Computing

```
  BoardSet.compare_elt
    [|[|(V, 2); (C, 2); (C, 3); (X, 0)|]; [|(V, 2); (H, 0); (H, 0); (V, 1)|];
      [|(V, 3); (X, 0); (C, 1); (V, 1)|]; [|(V, 3); (S, 0); (S, 0); (V, 0)|];
      [|(C, 0); (S, 0); (S, 0); (V, 0)|]|]
    [|[|(V, 2); (C, 2); (C, 3); (X, 0)|]; [|(V, 2); (H, 0); (H, 0); (V, 1)|];
      [|(V, 3); (X, 0); (C, 1); (V, 1)|]; [|(V, 3); (S, 0); (S, 0); (V, 0)|];
      [|(C, 0); (S, 0); (S, 0); (V, 0)|]|]
```

Correct value 0 1 pt

Computing

```
  BoardSet.compare_elt
    [|[|(C, 0); (V, 0); (V, 3); (V, 2)|]; [|(X, 0); (V, 0); (V, 3); (V, 2)|];
      [|(V, 1); (S, 0); (S, 0); (X, 0)|]; [|(V, 1); (S, 0); (S, 0); (C, 3)|];
      [|(C, 2); (C, 1); (H, 0); (H, 0)|]|]
```

```
    [|[|(C, 0); (V, 0); (V, 3); (X, 0)|]; [|(X, 0); (V, 0); (V, 3); (V, 2)|];
      [|(V, 1); (S, 0); (S, 0); (V, 2)|]; [|(V, 1); (S, 0); (S, 0); (C, 3)|];
      [|(C, 2); (C, 1); (H, 0); (H, 0)|]|]
```

Correct value 1 — 1 pt

```
Computing
  BoardSet.compare_elt
    [|[|(C, 2); (H, 0); (H, 0); (X, 0)|]; [|(S, 0); (S, 0); (V, 2); (C, 1)|];
      [|(S, 0); (S, 0); (V, 2); (V, 1)|]; [|(X, 0); (V, 3); (V, 0); (V, 1)|];
      [|(C, 0); (V, 3); (V, 0); (C, 3)|]|]
    [|[|(C, 2); (H, 0); (H, 0); (C, 1)|]; [|(S, 0); (S, 0); (V, 2); (X, 0)|];
      [|(S, 0); (S, 0); (V, 2); (V, 1)|]; [|(X, 0); (V, 3); (V, 0); (V, 1)|];
      [|(C, 0); (V, 3); (V, 0); (C, 3)|]|]
```

Correct value -1 — 1 pt

```
Computing
  BoardSet.compare_elt
    [|[|(X, 0); (C, 2); (C, 1); (V, 0)|]; [|(H, 0); (H, 0); (V, 1); (V, 0)|];
      [|(X, 0); (C, 3); (V, 1); (C, 0)|]; [|(V, 2); (S, 0); (S, 0); (V, 3)|];
      [|(V, 2); (S, 0); (S, 0); (V, 3)|]|]
    [|[|(X, 0); (C, 2); (C, 1); (V, 0)|]; [|(H, 0); (H, 0); (V, 1); (V, 0)|];
      [|(X, 0); (C, 3); (V, 1); (C, 0)|]; [|(V, 2); (S, 0); (S, 0); (V, 3)|];
      [|(V, 2); (S, 0); (S, 0); (V, 3)|]|]
```

Correct value 0 — 1 pt

```
Computing
  BoardSet.compare_elt
    [|[|(V, 3); (C, 0); (V, 1); (X, 0)|]; [|(V, 3); (C, 3); (V, 1); (C, 1)|];
      [|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (V, 2); (V, 0)|];
      [|(C, 2); (X, 0); (V, 2); (V, 0)|]|]
    [|[|(V, 3); (C, 0); (V, 1); (X, 0)|]; [|(V, 3); (C, 3); (V, 1); (C, 1)|];
      [|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (V, 2); (V, 0)|];
      [|(X, 0); (C, 2); (V, 2); (V, 0)|]|]
```

Correct value 1 — 1 pt

```
Computing
  BoardSet.compare_elt
    [|[|(H, 0); (H, 0); (V, 0); (C, 3)|]; [|(C, 0); (X, 0); (V, 0); (V, 3)|];
      [|(V, 1); (X, 0); (C, 2); (V, 3)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(C, 1); (S, 0); (S, 0); (V, 2)|]|]
    [|[|(H, 0); (H, 0); (V, 0); (C, 3)|]; [|(X, 0); (C, 0); (V, 0); (V, 3)|];
      [|(V, 1); (X, 0); (C, 2); (V, 3)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(C, 1); (S, 0); (S, 0); (V, 2)|]|]
```

Correct value 1 — 1 pt

```
Computing
  BoardSet.compare_elt
    [|[|(V, 1); (C, 3); (C, 1); (V, 3)|]; [|(V, 1); (H, 0); (H, 0); (V, 3)|];
      [|(V, 2); (X, 0); (C, 2); (V, 0)|]; [|(V, 2); (S, 0); (S, 0); (V, 0)|];
      [|(C, 0); (S, 0); (S, 0); (X, 0)|]|]
    [|[|(V, 2); (S, 0); (S, 0); (C, 1)|]; [|(V, 2); (S, 0); (S, 0); (X, 0)|];
      [|(V, 0); (C, 3); (V, 3); (C, 0)|]; [|(V, 0); (C, 2); (V, 3); (V, 1)|];
      [|(H, 0); (H, 0); (X, 0); (V, 1)|]|]
```

Correct value -1 — 1 pt

```
Computing
```



```
    [|[|(X, 0); (H, 0); (H, 0); (X, 0)|]; [|(V, 0); (C, 3); (V, 3); (C, 1)|];
      [|(V, 0); (C, 2); (V, 3); (V, 2)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(V, 1); (S, 0); (S, 0); (C, 0)|]|]
```

Correct value 1 — 1 pt

```
Computing
  BoardSet.compare_elt
    [|[|(S, 0); (S, 0); (C, 2); (V, 0)|]; [|(S, 0); (S, 0); (X, 0); (V, 0)|];
      [|(V, 2); (X, 0); (V, 1); (V, 3)|]; [|(V, 2); (C, 1); (V, 1); (V, 3)|];
      [|(C, 3); (C, 0); (H, 0); (H, 0)|]|]
    [|[|(S, 0); (S, 0); (X, 0); (V, 0)|]; [|(S, 0); (S, 0); (C, 2); (V, 0)|];
      [|(V, 2); (X, 0); (V, 1); (V, 3)|]; [|(V, 2); (C, 1); (V, 1); (V, 3)|];
      [|(C, 3); (C, 0); (H, 0); (H, 0)|]|]
```

Correct value 1 — 1 pt

```
Computing
  BoardSet.compare_elt
    [|[|(V, 1); (H, 0); (H, 0); (C, 3)|]; [|(V, 1); (S, 0); (S, 0); (C, 0)|];
      [|(V, 0); (S, 0); (S, 0); (X, 0)|]; [|(V, 0); (C, 2); (V, 2); (V, 3)|];
      [|(X, 0); (C, 1); (V, 2); (V, 3)|]|]
    [|[|(V, 3); (X, 0); (H, 0); (H, 0)|]; [|(V, 3); (V, 1); (C, 0); (V, 2)|];
      [|(V, 0); (V, 1); (C, 3); (V, 2)|]; [|(V, 0); (S, 0); (S, 0); (C, 1)|];
      [|(C, 2); (S, 0); (S, 0); (X, 0)|]|]
```

Correct value -1 — 1 pt

```
Computing
  BoardSet.compare_elt
    [|[|(C, 2); (S, 0); (S, 0); (C, 3)|]; [|(X, 0); (S, 0); (S, 0); (V, 1)|];
      [|(V, 2); (H, 0); (H, 0); (V, 1)|]; [|(V, 2); (V, 3); (X, 0); (V, 0)|];
      [|(C, 0); (V, 3); (C, 1); (V, 0)|]|]
    [|[|(C, 2); (S, 0); (S, 0); (C, 3)|]; [|(X, 0); (S, 0); (S, 0); (V, 1)|];
      [|(V, 2); (H, 0); (H, 0); (V, 1)|]; [|(V, 2); (V, 3); (X, 0); (V, 0)|];
      [|(C, 0); (V, 3); (C, 1); (V, 0)|]|]
```

Correct value 0 — 1 pt

```
Computing
```

```
  BoardSet.compare_elt
    [|[|(V, 3); (V, 1); (C, 3); (C, 1)|]; [|(V, 3); (V, 1); (H, 0); (H, 0)|];
      [|(V, 0); (X, 0); (C, 2); (V, 2)|]; [|(V, 0); (S, 0); (S, 0); (V, 2)|];
      [|(X, 0); (S, 0); (S, 0); (C, 0)|]|]
    [|[|(C, 1); (H, 0); (H, 0); (C, 0)|]; [|(X, 0); (V, 3); (V, 0); (V, 1)|];
      [|(C, 2); (V, 3); (V, 0); (V, 1)|]; [|(C, 3); (S, 0); (S, 0); (V, 2)|];
      [|(X, 0); (S, 0); (S, 0); (V, 2)|]|]
```

Correct value -1 — 1 pt

Computing

```
  BoardSet.compare_elt
    [|[|(C, 2); (C, 3); (C, 1); (X, 0)|]; [|(S, 0); (S, 0); (C, 0); (V, 1)|];
      [|(S, 0); (S, 0); (V, 2); (V, 1)|]; [|(V, 3); (V, 0); (V, 2); (X, 0)|];
      [|(V, 3); (V, 0); (H, 0); (H, 0)|]|]
    [|[|(C, 2); (C, 3); (C, 1); (X, 0)|]; [|(S, 0); (S, 0); (C, 0); (V, 1)|];
      [|(S, 0); (S, 0); (V, 2); (V, 1)|]; [|(V, 3); (V, 0); (V, 2); (X, 0)|];
      [|(V, 3); (V, 0); (H, 0); (H, 0)|]|]
```

Correct value 0 — 1 pt

Computing

```
  BoardSet.compare_elt
    [|[|(C, 2); (V, 1); (C, 1); (C, 3)|]; [|(V, 3); (V, 1); (X, 0); (X, 0)|];
      [|(V, 3); (H, 0); (H, 0); (V, 2)|]; [|(V, 0); (S, 0); (S, 0); (V, 2)|];
      [|(V, 0); (S, 0); (S, 0); (C, 0)|]|]
    [|[|(C, 2); (V, 1); (X, 0); (C, 3)|]; [|(V, 3); (V, 1); (C, 1); (X, 0)|];
      [|(V, 3); (H, 0); (H, 0); (V, 2)|]; [|(V, 0); (S, 0); (S, 0); (V, 2)|];
      [|(V, 0); (S, 0); (S, 0); (C, 0)|]|]
```

Correct value 1 — 1 pt

v Exercise 17: `compare` — Completed, 30 pts

Found `BoardSet.compare_elt` with compatible type.

Computing

```
  BoardSet.compare_elt
    [|[|(V, 3); (V, 2); (C, 1); (V, 1)|]; [|(V, 3); (V, 2); (X, 0); (V, 1)|];
      [|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (V, 0); (C, 3)|];
      [|(X, 0); (C, 2); (V, 0); (C, 0)|]|]
    [|[|(V, 3); (V, 2); (C, 1); (V, 1)|]; [|(V, 3); (V, 2); (X, 0); (V, 1)|];
      [|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (V, 0); (C, 3)|];
      [|(C, 2); (X, 0); (V, 0); (C, 0)|]|]
```

Correct value -1 — 1 pt

Checking the number of array accesses.

Correct value 44 — 1 pt

Computing

```
  BoardSet.compare_elt
    [|[|(V, 3); (S, 0); (S, 0); (C, 0)|]; [|(V, 3); (S, 0); (S, 0); (X, 0)|];
      [|(V, 0); (V, 1); (H, 0); (H, 0)|]; [|(V, 0); (V, 1); (C, 3); (V, 2)|];
      [|(C, 2); (X, 0); (C, 1); (V, 2)|]|]
    [|[|(V, 3); (S, 0); (S, 0); (C, 0)|]; [|(V, 3); (S, 0); (S, 0); (X, 0)|];
      [|(V, 0); (V, 1); (H, 0); (H, 0)|]; [|(V, 0); (V, 1); (C, 3); (V, 2)|];
      [|(C, 2); (X, 0); (C, 1); (V, 2)|]|]
```

Correct value 0 — 1 pt

Checking the number of array accesses.


```
    [|[|(V, 3); (X, 0); (C, 2); (V, 0)|]; [|(V, 3); (H, 0); (H, 0); (V, 0)|];
      [|(V, 1); (C, 0); (C, 1); (V, 2)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(C, 3); (S, 0); (S, 0); (X, 0)|]|]
    [|[|(V, 3); (X, 0); (C, 2); (V, 0)|]; [|(V, 3); (H, 0); (H, 0); (V, 0)|];
      [|(V, 1); (C, 0); (C, 1); (X, 0)|]; [|(V, 1); (S, 0); (S, 0); (V, 2)|];
      [|(C, 3); (S, 0); (S, 0); (V, 2)|]|]
```

Correct value 1 — 1 pt

Checking the number of array accesses.

Correct value 30 — 1 pt

Computing

```
  BoardSet.compare_elt
    [|[|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (V, 3); (C, 3)|];
      [|(X, 0); (C, 1); (V, 3); (V, 1)|]; [|(V, 2); (C, 2); (V, 0); (V, 1)|];
      [|(V, 2); (C, 0); (V, 0); (X, 0)|]|]
    [|[|(C, 1); (C, 2); (V, 0); (C, 3)|]; [|(H, 0); (H, 0); (V, 0); (V, 1)|];
      [|(V, 2); (X, 0); (X, 0); (V, 1)|]; [|(V, 2); (S, 0); (S, 0); (V, 3)|];
      [|(C, 0); (S, 0); (S, 0); (V, 3)|]|]
```

Correct value 1 — 1 pt

Checking the number of array accesses.

Correct value 4 — 1 pt

Computing

```
  BoardSet.compare_elt
    [|[|(X, 0); (C, 2); (V, 0); (V, 1)|]; [|(H, 0); (H, 0); (V, 0); (V, 1)|];
      [|(V, 3); (X, 0); (C, 0); (C, 3)|]; [|(V, 3); (S, 0); (S, 0); (V, 2)|];
      [|(C, 1); (S, 0); (S, 0); (V, 2)|]|]
    [|[|(X, 0); (C, 2); (V, 0); (V, 1)|]; [|(H, 0); (H, 0); (V, 0); (V, 1)|];
      [|(V, 3); (C, 0); (X, 0); (C, 3)|]; [|(V, 3); (S, 0); (S, 0); (V, 2)|];
      [|(C, 1); (S, 0); (S, 0); (V, 2)|]|]
```

Correct value -1 — 1 pt

Checking the number of array accesses.

Correct value 26 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(V, 1); (C, 3); (X, 0); (C, 1)|]; [|(V, 1); (C, 2); (V, 2); (V, 3)|];
    [|(S, 0); (S, 0); (V, 2); (V, 3)|]; [|(S, 0); (S, 0); (X, 0); (V, 0)|];
    [|(C, 0); (H, 0); (H, 0); (V, 0)|]|]
  [|[|(V, 1); (C, 3); (X, 0); (C, 1)|]; [|(V, 1); (C, 2); (V, 2); (V, 3)|];
    [|(S, 0); (S, 0); (V, 2); (V, 3)|]; [|(S, 0); (S, 0); (X, 0); (V, 0)|];
    [|(C, 0); (H, 0); (H, 0); (V, 0)|]|]
```

Correct value 0 1 pt

Checking the number of array accesses.

Correct value 50 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(C, 2); (C, 3); (X, 0); (V, 1)|]; [|(H, 0); (H, 0); (V, 0); (V, 1)|];
    [|(V, 2); (C, 1); (V, 0); (C, 0)|]; [|(V, 2); (S, 0); (S, 0); (V, 3)|];
    [|(X, 0); (S, 0); (S, 0); (V, 3)|]|]
  [|[|(C, 2); (S, 0); (S, 0); (V, 3)|]; [|(V, 1); (S, 0); (S, 0); (V, 3)|];
    [|(V, 1); (X, 0); (V, 2); (X, 0)|]; [|(C, 0); (C, 3); (V, 2); (V, 0)|];
    [|(C, 1); (H, 0); (H, 0); (V, 0)|]|]
```

Correct value -1 1 pt

Checking the number of array accesses.

Correct value 6 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(V, 0); (C, 0); (H, 0); (H, 0)|]; [|(V, 0); (S, 0); (S, 0); (X, 0)|];
    [|(C, 2); (S, 0); (S, 0); (C, 3)|]; [|(V, 3); (V, 1); (V, 2); (X, 0)|];
    [|(V, 3); (V, 1); (V, 2); (C, 1)|]|]
  [|[|(V, 0); (C, 0); (H, 0); (H, 0)|]; [|(V, 0); (S, 0); (S, 0); (C, 3)|];
    [|(C, 2); (S, 0); (S, 0); (X, 0)|]; [|(V, 3); (V, 1); (V, 2); (X, 0)|];
    [|(V, 3); (V, 1); (V, 2); (C, 1)|]|]
```

Correct value -1 1 pt

Checking the number of array accesses.

Correct value 20 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(V, 3); (C, 0); (V, 1); (V, 0)|]; [|(V, 3); (C, 2); (V, 1); (V, 0)|];
    [|(C, 3); (H, 0); (H, 0); (C, 1)|]; [|(V, 2); (S, 0); (S, 0); (X, 0)|];
    [|(V, 2); (S, 0); (S, 0); (X, 0)|]|]
  [|[|(V, 3); (C, 0); (V, 1); (V, 0)|]; [|(V, 3); (C, 2); (V, 1); (V, 0)|];
    [|(C, 3); (H, 0); (H, 0); (C, 1)|]; [|(V, 2); (X, 0); (S, 0); (S, 0)|];
    [|(V, 2); (X, 0); (S, 0); (S, 0)|]|]
```

Correct value 1 1 pt

Checking the number of array accesses.

Correct value 36 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(V, 3); (V, 2); (C, 1); (C, 2)|]; [|(V, 3); (V, 2); (V, 0); (V, 1)|];
```



```
    [|(H, 0); (H, 0); (X, 0); (V, 3)|]; [|(C, 1); (S, 0); (S, 0); (V, 0)|];
    [|(C, 3); (S, 0); (S, 0); (V, 0)|]|]
```

Correct value 1 1 pt

Checking the number of array accesses.

Correct value 4 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(C, 1); (X, 0); (C, 2); (C, 0)|]; [|(V, 1); (H, 0); (H, 0); (V, 3)|];
    [|(V, 1); (X, 0); (C, 3); (V, 3)|]; [|(V, 2); (S, 0); (S, 0); (V, 0)|];
    [|(V, 2); (S, 0); (S, 0); (V, 0)|]|]
  [|[|(C, 1); (X, 0); (C, 2); (C, 0)|]; [|(V, 1); (H, 0); (H, 0); (V, 3)|];
    [|(V, 1); (C, 3); (X, 0); (V, 3)|]; [|(V, 2); (S, 0); (S, 0); (V, 0)|];
    [|(V, 2); (S, 0); (S, 0); (V, 0)|]|]
```

Correct value -1 1 pt

Checking the number of array accesses.

Correct value 26 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (X, 0); (C, 0)|];
    [|(V, 1); (V, 0); (V, 3); (V, 2)|]; [|(V, 1); (V, 0); (V, 3); (V, 2)|];
    [|(C, 3); (C, 1); (C, 2); (X, 0)|]|]
  [|[|(S, 0); (S, 0); (H, 0); (H, 0)|]; [|(S, 0); (S, 0); (X, 0); (C, 0)|];
    [|(V, 1); (V, 0); (V, 3); (V, 2)|]; [|(V, 1); (V, 0); (V, 3); (V, 2)|];
    [|(C, 3); (C, 1); (C, 2); (X, 0)|]|]
```

Correct value 0 1 pt

Checking the number of array accesses.

Correct value 50 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(V, 2); (V, 3); (V, 1); (V, 0)|]; [|(V, 2); (V, 3); (V, 1); (V, 0)|];
    [|(C, 0); (C, 3); (H, 0); (H, 0)|]; [|(X, 0); (S, 0); (S, 0); (X, 0)|];
    [|(C, 1); (S, 0); (S, 0); (C, 2)|]|]
  [|[|(V, 2); (V, 3); (V, 1); (V, 0)|]; [|(V, 2); (V, 3); (V, 1); (V, 0)|];
    [|(C, 0); (C, 3); (H, 0); (H, 0)|]; [|(X, 0); (S, 0); (S, 0); (C, 2)|];
    [|(C, 1); (S, 0); (S, 0); (X, 0)|]|]
```

Correct value -1 — 1 pt

Checking the number of array accesses.

Correct value 40 — 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(V, 3); (V, 0); (X, 0); (C, 3)|]; [|(V, 3); (V, 0); (H, 0); (H, 0)|];
    [|(C, 1); (C, 0); (X, 0); (V, 1)|]; [|(V, 2); (S, 0); (S, 0); (V, 1)|];
    [|(V, 2); (S, 0); (S, 0); (C, 2)|]|]
  [|[|(X, 0); (X, 0); (C, 1); (V, 3)|]; [|(C, 2); (S, 0); (S, 0); (V, 3)|];
    [|(V, 1); (S, 0); (S, 0); (V, 0)|]; [|(V, 1); (V, 2); (C, 0); (V, 0)|];
    [|(C, 3); (V, 2); (H, 0); (H, 0)|]|]
```

Correct value 1 — 1 pt

Checking the number of array accesses.

Correct value 4 — 1 pt

Computing

```
BoardSet.compare_elt
  [|[|(C, 2); (S, 0); (C, 0)|]; [|(C, 1); (S, 0); (S, 0); (X, 0)|];
    [|(C, 3); (X, 0); (H, 0); (H, 0)|]; [|(V, 3); (V, 0); (V, 1); (V, 2)|];
    [|(V, 3); (V, 0); (V, 1); (V, 2)|]|]
  [|[|(C, 2); (S, 0); (S, 0); (X, 0)|]; [|(C, 1); (S, 0); (S, 0); (C, 0)|];
    [|(C, 3); (X, 0); (H, 0); (H, 0)|]; [|(V, 3); (V, 0); (V, 1); (V, 2)|];
    [|(V, 3); (V, 0); (V, 1); (V, 2)|]|]
```

Correct value 1 — 1 pt

Checking the number of array accesses.

Correct value 10 — 1 pt

v Exercise 18: `solve_klotski` — Completed, 10 pts

Found `solve_klotski` with compatible type.

Computing `solve initial_board_simpler`

Your solver is correct. Congratulations ! — 10 pts