COMBINATORIAL AND POSITIONAL GAME THEORY IN LEAN

YVES JÄCKLE

ABSTRACT. *This article is meant as a reading guide to an ongoing formalization project. We also use it to compare our formalization to existing ones. We thus warn the reader that the formalism may evolve in the future, making this article outdated.*
We present a formalization of classic results from combinatorial and positional game theory. We formalize the notions of games and strategies, as well as those of "strategy stealing", positional games, and "pairing strategies". We illustrate applications of these notions to the games of Pick-up-bricks, Chomp and Tic-tac-toe. Throughout the article, we compare the formalism to related work.

Lean [1] is a interactive theorem prover and programming language first released by Leonardo de Moura in 2013. Its type theory is a variant of the *calculus of inductive constructions*, similar to those of Agda [2] or Coq [3], and its kernel is being formally verified in ongoing work by Mario Carneiro [4]. Lean hosts Mathlib [5], an ever-growing library providing unified formal foundations to many fields of mathematics. A list of successful formalization projects using Lean may be found at [6].

We've formalized content from combinatorial and positional game theory, based on the presentations given in textbooks [7] and [8]. Throughout the article, the game theory we refer to is the one modeling two-player turn-based games, which terminate (if they do) in win/loose/draw states, rather then in numeric payoffs for the players.

Combinatorial game theory was already given formal foundations in Lean in work initiated by Isabel Longbottom [9] in 2019. Currently, Mathlib contains formal foundations of combinatorial game theory (as well as applications to the games of Nim and Domineering) at [10], with main contributors being: Reid Barton, Mario Carneiro, Markus Himmel, Isabel Longbottom, Kim Morrison, Apurva Nakade, Violeta Hernández Palacios and Fox Thomson. Recently, Sven Manthe has given further foundations in [14]. In Isabelle/HOL [12], foundations were given by Sebastiaan J. C. Joosten [11] in 2021, based on the approach of Christoph Dittmann [15]. The blog post [13] of 2013 develops a formalism in Coq.

We are not aware further published work on formal combinatorial game theory.
We will present the different formalisms and their ways of treating common informal concepts throughout this article, when the concept in question is being discussed.
Our code is available at https://github.com/Happyves/Lean_Games in the folders "gameLib" and "games". The code runs on Lean version 4.5.

Date: November 6, 2024.
MSC2020: Primary 00A05, Secondary 00A66.

Fundamentals

First, we'll discuss the very fundamentals of the formalism: the notions of games, strategies, and of termination. In our formalism, games are defined via:

```
structure Game_World (α β : Type _) where
  init_game_state : α
  fst_win_states : List β →  Prop
  snd_win_states : List β → Prop
  fst_legal : List β → (β → Prop)
  snd_legal : List β → (β → Prop)
  fst_transition : List β → β → α
  snd_transition : List β → β → α
```

Provided a type α for the state of the game and a type β of moves the two players may play at each turn, a game is defined by the following data:

- an initial state
- two predicates `fst_win_states` and `snd_win_states` that decide, given the history of moves played so far - in the form of a list of terms of the move type - (and implicitly the initial state), whether the game is won by the corresponding player.
- two predicates `fst_legal` and `snd_legal` that decide, given the history of moves played so far, whether the next move to be played is allowed to be played by the game's rules.
- two transition functions `fst_transition` and `snd_transition` that return the state of the game, provided the history of moves and the last move separately.

We define the similar notions of `Symm_Game_World`, where the legality predicates and transition functions are the same for both players, and that of `Game_World_wDraw`, in which there is an additional predicate deciding if the game has reached a draw.

To illustrate the formalism, we'll apply it to the game of *Pick-up-bricks* [7]. This game is played with a stack of bricks as follows: at each turn, the players may pick one or two bricks from the stack, and the player to take the last brick(s) of the stack wins the game. One can show that there is a winning strategy for the second player, if the initial number of bricks is divisible by 3. Indeed, the second player may act as follows: if the first player, takes one brick, take two, and if they took tow, take one. This way, after each round, when both player took their turn, 3 bricks where taken, hence the number of bricks on the stack remains divisible by 3. In fact, the stack of bricks is divisible by 3 only after the second players turn. So the last brick taken must have been taken by the second player, as 3 divides 0.

Formally, we define Pick-up-bricks as follows:

```
variable (init_bricks : ℕ)

def bricks_from_ini_hist (ini : ℕ) (hist : List ℕ) := ini - hist.sum
def bricks_from_ini_hist_act (ini : ℕ) (hist : List ℕ) (act : ℕ) := ini -
    hist.sum - act

def PickUpBricks : Symm_Game_World ℕ ℕ where
  init_game_state := init_bricks
  fst_win_states := (fun hist => Turn_fst hist.length ∧ bricks_from_ini_hist
    init_bricks hist = 0)
  snd_win_states := (fun hist => Turn_snd hist.length ∧ bricks_from_ini_hist
    init_bricks hist = 0)
  transition :=  bricks_from_ini_hist_act init_bricks
  law := fun _ act =>  act = 1 ∨ act = 2
```

Here, the state of the game is the number of bricks on the stack, and the moves are the number of bricks taken by the players. The initial number of bricks on the stack is given as a variable. The game is won by a player if there are no bricks left on the stack at the end of their turn. We note that `Turn_fst` indicate that the turn was the first players, and simply reduces to saying that the turn is odd, while `Turn_snd` indicate that the turn was the second players, and simply reduces to saying that the turn is even. Here, a move, ie. the number of brick picked, is legal if its 1 or 2. In more complex games, the legality of the move may depend on the previous moves (not simply the state: consider the case of castling in chess), which our formalism may express.

We now formalize the notion of strategy:

```
def Game_World.fStrategy (g : Game_World α β) := (hist : List β) →
    (Turn_fst (hist.length+1)) → (g.hist_legal hist) →
    { act : β // g.fst_legal hist act}

def Game_World.sStrategy (g : Game_World α β) := (hist : List β) →
    (Turn_snd (hist.length+1)) → (g.hist_legal hist) →
    { act : β // g.snd_legal hist act}
```

where we defined:

```
inductive Game_World.hist_legal (g : Game_World α β)  : List β → Prop
| nil : g.hist_legal []
| cons (l : List β) (act : β) : (if Turn_fst (l.length + 1)
                                then g.fst_legal l act
                                else g.snd_legal l act) →
                                     g.hist_legal l →
                                     g.hist_legal (act :: l)
```

A strategy will be a function mapping a history of previous moves to a move to be played. In our definitions, the strategy may access the initial state via the provided `Game_World`. Its inputs are then the history of moves played so far, a proof that it currently is the respective players turn, and a proof that the input history is made up of legal moves. It returns a subtype made of the played move and a proof that this move is legal. This definition avoids having to let strategies return dummy values for histories not made up of legal moves. We are currently experimenting with having strategies also require that the input history doesn't satisfy the winning predicates, as the game will be considered finished at that stage. In the current formalism however, strategies must play dummy values for histories in which we consider the game to be over, and we'll see the consequences that arise due to this further on in our discussion.

We briefly discuss what it means for a history to be legal. This predicate on histories must be built using its two constructors: empty histories are legal, and a history is legal if its tail is, and its head - the last move - is legal wrt. legality predicate of the player who's tun it is.
Though we don't reproduce the code here, we may define, given strategies for either player, the history given a turn, `Game_World.hist_on_turn`. It is defined recursively: on turn 0 the history is empty, and on turn $t+1$, we prepend the move returned by the strategy of the current player, queried on the history on turn t, to the history on turn t.

We define an additional structure for convenience, it which we glue the strategies of the players to the other data of the game:

```
structure Game (α β : Type _) extends Game_World α β where
  fst_strat : toGame_World.fStrategy
  snd_strat : toGame_World.sStrategy
```

Next, we define what it means for a game to terminate:

```
inductive Game_World.Turn_isWL (g : Game_World α β)
  (f_strat : g.fStrategy) (s_strat : g.sStrategy) (turn : ℕ) : Prop where
  | wf : g.fst_win_states (g.hist_on_turn f_strat s_strat turn) →
    g.Turn_isWL f_strat s_strat turn
  | ws : g.snd_win_states  (g.hist_on_turn f_strat s_strat turn) →
    g.Turn_isWL f_strat s_strat turn

def Game_World.isWL (g : Game_World α β) : Prop :=
  ∀ (f_strat : g.fStrategy), ∀ (s_strat : g.sStrategy),
  ∃ turn, g.Turn_isWL f_strat s_strat turn
```

The first definition above expresses that a turn is terminal, which may occur in two ways: either the history on that turn satisfies the winning predicate of the first player or that of the second (without further assumption, it can do both simultaneously).
The second definition expresses that a game terminates: it reaches a winning state for one of the players, not matter how they play.

Turns that aren't winning are said to be neutral:

```
def Game_World.state_on_turn_neutral (g : Game_World α β)
  (f_strat : g.fStrategy) (s_strat : g.sStrategy) (turn : ℕ) : Prop :=
  ¬ g.Turn_isWL f_strat s_strat turn

def Game.state_on_turn_neutral (g : Game α β) (turn : ℕ) : Prop :=
  g.toGame_World.state_on_turn_neutral g.fst_strat g.snd_strat turn
```

With these definitions in place, we may now defined what it means for a game to be won by either player:

```
def Game.fst_win (g : Game α β) : Prop :=
  ∃ turn : ℕ, g.fst_win_states (g.hist_on_turn turn) ∧ (∀ t < turn,
    g.state_on_turn_neutral t)

def Game.snd_win (g : Game α β) : Prop :=
  ∃ turn : ℕ, g.snd_win_states (g.hist_on_turn turn) ∧ (∀ t < turn,
    g.state_on_turn_neutral t)
```

In words: there comes a turn in which the winning predicate is satisfied by the history of moves, and none of the previous turns satisfied these predicates.
Finally, we may define the notions of winning strategies:

```
def Game_World.is_fst_win (g : Game_World α β) : Prop :=
  ∃ ws : g.fStrategy, ∀ snd_s : g.sStrategy,
  ({g with fst_strat := ws, snd_strat := snd_s} : Game α β).fst_win

def Game_World.is_snd_win (g : Game_World α β) : Prop :=
  ∃ ws : g.sStrategy, ∀ snd_s : g.fStrategy,
  ({g with snd_strat := ws, fst_strat := snd_s} : Game α β).snd_win
```

In words: the corresponding player has a strategy that beats all possible strategies of the other player. We invite the reader to see these definitions applied to the game of Pick-up-bricks, in the file `games > PickUpBricks`.

We will now discuss the formalisms used in related work.

First, we discuss the formalism from the blog post [13], implemented in Coq. In this formalism, players "Left" and "Right" play games defined via:

```
inductive player where
| left | right

structure combinatorial_game where
  position : Type
  moves : player → position → List position
  valid_move := fun (next current : position) => ∃ s : player, next ∈ (moves
    s current)
  finite_game : WellFounded valid_move
```

Here, "position" denotes the type of the state of the game (our α). "moves" returns, given the current player and the current position, a list of positions that are available as legal outcomes of the players move (our `fst_legal` and `snd_legal`). Note that in general, using only a state instead of a history of moves doesn't affect the expressivity of the formalism, but rather its ease of use: we may consider the history so far to *be* the state of the game, and the actual state may be obtain via transition functions, and is purely accessory. Before moving on, note that restricting the next positions to be in a list implies that there may only be finitely many of them, a restriction not present in our formalism. Next, a relation "valid_move" is defined, which states that a position may legally follow from the previous. It is then required that the relation be well-founded, aka. there may not be infinite descending chains for this relations, or in this context, no infinite sequences of positions that may follow from each other. We do not require this in our formalism, though theorems such as Zermelo will require assumptions on the games that amount to this. A general notion of a game is defined via:

```
inductive game where
| mk (left_moves : List game) (right_moves : List game)

def game_as_cg : combinatorial_game :=
  {position := game
   moves := fun s g =>
      match s, g with
      | .left, .mk left_moves _ => left_moves
      | .right, .mk right_moves _ => right_moves
   finite_game := sorry
  }
```

Here, "game" can be pictured as a tree with two kinds on children, one for each player: "moving to a child" corresponds to the player who's turn it is making a move that places the game in the position that is this child. Once we reach a node with no children, the game has ended. In the blog post, it is shown that this instance of the "match" relation is indeed well-founded.

The notion of a game being won is formalized by:

```
def other (s : player) : player :=
  match s with
    | .left => .right
    | .right => .left

inductive Match (cg : combinatorial_game) :
  ∀ (first winner : player), List (cg.position) → Prop :=
  | Match_end : ∀ pl pos,
                cg.moves pl pos = [] →
                Match cg pl (other pl) [pos]
  | Match_move : ∀ pl winner pos next m,
                  next ∈ (cg.moves pl pos) →
                  Match cg (other pl) winner (next :: m) →
                  Match cg pl winner (pos :: next :: m)
```

The above predicate expresses that the game, when played such that `first` goes first, is won by player `winner`, with the games complete history of positions being the list that is the final input to the predicate. To prove this property, we must find a player and a position from which not more moves may be played, at which stage the other player is declared the winner (`Match_end`). If this stage can be reached from another, the predicate is also satisfied when prepending the current position to the one leading to victory (`Match_move`). The notions of strategies and of winning strategies are not developed in this formalism.

Next, we discuss Joosten's formalism [11], which is in the same vein as that of the above blog post and of ours. The central notion is similar to our notion of a history of moves, in the form of lists where even moves correspond to those of the player "Even", and the odd ones to those of the player "Odd". Roughly speaking, games (`GSgame`) are defined by a set of sequences of moves that correspond to a win of the first player, similar to our predicate `fst_win_states`. A difference to our formalism is that that the sequences defining winning games must have the same fixed (but possibly infinite) length. As can be seen in `strategy-winning-by-Odd`, sequences of moves of the game's length not in this set will be considered a win for the second player. As in our formalism, strategies are defined as functions mapping lists of previous moves to next moves. A key difference is that these are total functions, that aren't restricted to legal histories of moves. Winning strategies are defined as those for which, for all strategies of the other player, the game reaches a winning state for the current player.

Finally, we discuss the formalism developed by Longbottom [9] and adopted by Mathlib [10]. Here, we start with the notion of a pre-game:

```
inductive PGame : Type (u + 1)
  | mk : ∀ α β : Type u, (α → PGame) → (β → PGame) → PGame
```

It generalises the `game` from [13] as follows: rather then having a move consisting of choosing a next state among a list of available child states, we select a move among as a term of a type (α for the left player, β for the right) and use the functions from the constructor to move to the

next stage in the game. Since α and β aren't parameters, the type of the moves may change from turn to turn. Next, a relation between games is defined:

```
inductive IsOption : PGame → PGame → Prop
  | moveLeft {x : PGame} (i : x.LeftMoves) : IsOption (x.moveLeft i) x
  | moveRight {x : PGame} (i : x.RightMoves) : IsOption (x.moveRight i) x
```

It states that the first game may be considered an outcome of the second by a single turn, and the constructors instruct on which player made which move. Perhaps surprisingly, this relation is well-founded, as is shown in `SetTheory.PGame.wf_isOption`. From this relation, the relation `SetTheory.PGame.le` is eventually defined, which is used to express the notions of winning strategies. It's perhaps best understood from:

```
theorem zero_le {x : PGame} :
    0 ≤ x ↔ ∀ j, ∃ i, 0 ≤ (x.moveRight j).moveLeft i := sorry
theorem le_zero {x : PGame} :
    x ≤ 0 ↔ ∀ i, ∃ j, (x.moveLeft i).moveRight j ≤ 0 := sorry
```

As before, we consider a game to be lost by a player if they have no moves left to play, in the sense that the corresponding move type in the games constructor is `PEmpty`. By letting 0 denote an empty, and hence losing-position-game, the above theorems state that in a game in relation to 0, the second player (either Left or Right), can maintain that relation, no matter what the first plays. Then, intuitively speaking, since by well-foundedness there may not be an infinitely long sequence of such pairs of moves, the game ends, and it can't have been the first player to make the last move, since by the above theorems, the second player has an answer, hence a move, to follow with.

Notes

We'd like to comment on the expressivity of our formalism.

- Games in which players make moves of different types is not a simple curiosity: we refer to chapter 3 of [8] on biased games, where players makes moves best described by different types. We can model games such as these as follows: we use, as type for moves the type below, and legal predicates in the pattern described below that.

  ```
  inductive Moves (α β : Type _) where
  | ofFirst (val : α)
  | ofSecond (val : β)

  def fst_legal {actual : List (Moves α β) → α → Prop}
    (hist : List (Moves α β)) (act : Moves α β) : Prop :=
      match act with
      | .ofFirst val => actual hist val
      | .ofSecond _ => False
  ```

- It is possible to model games in which the types of actions to be played depends on the turn, for example by using the following type for moves, and having legal predicates in the pattern described below.

```
inductive Moves (fam : Nat → Type _) where
| mk (idx : Nat) (val : fam idx)

def leg {fam : Nat → Type _}
    {actual : (hist : List (Moves fam)) → fam hist.length → Prop}
    (hist : List (Moves fam)) (act : (Moves fam)) : Prop :=
      match act with
      | .mk i val =>
          if H : i = hist.length
          then actual hist (by rw [← H] ; exact val)
          else False
```

Zermelo's theorem

Zermelo's theorem roughly states that games that terminate in either a win or a loss for the players, have a winning strategy for one of the players. The gist of the proof is that if at a stage in the game, we can determine for each playable move, if the game states after these moves have winning strategies, then we can determine which player has a winning strategy at the current stage: if there is a move such that the current player has a winning strategy from the stage thereon, they may play that move, and according to the strategy afterwards, and this constitutes a winning strategy for the current stage ; if on the other hand, no matter the move, the other player has a winning strategy in each outcome, then the other player has a winning strategy at the current stage (let the current player make a move, and beat them with the winning strategy for the game after that move).

Our statement of Zermelo's theorem is:

```
lemma Game_World.Zermelo [DecidableEq β] (g : Game_World α β)
  (hgw : g.isWL) (hgp : g.playable) (hgn : g.coherent_end) :
  g.has_WL
```

Its first assumption was already explained (`isWL`: the game eventually reaches a winning state, no matter the strategies played). We now discuss the two others, starting with:

```
def Game_World.playable (g : Game_World α β) : Prop :=
  ∀ hist : List β, g.hist_legal hist →
    ((Turn_fst (List.length hist + 1) → ∃ act : β, g.fst_legal hist act)
    ∧ (Turn_snd (List.length hist + 1) → ∃ act : β, g.snd_legal hist act))
```

We call a game *playable* if for any history made up of legal moves, there exists a move that may legally be played next. Note that since strategies are required to return moves even when the game has ended, one has to make sure, when defining a game we wish to be *playable*, to have legal dummy moves to play, once the game is over, morally speaking. We'll see this requirement in action in the game of Chomp, discussed next section. Next, we have:

```
structure Game_World.coherent_end (g : Game_World α β) : Prop where
  em : ∀ hist, g.hist_legal hist →
            ¬ (g.fst_win_states hist ∧ g.snd_win_states hist)
  fst : ∀ hist, g.hist_legal hist → (g.fst_win_states hist →
          ∀ act, ((Turn_fst (hist.length+1) → g.fst_legal hist act)
            ∧ (Turn_snd (hist.length+1) → g.snd_legal hist act)) →
                g.fst_win_states (act :: hist))
  snd : ∀ hist, g.hist_legal hist → (g.snd_win_states hist →
          ∀ act, ((Turn_fst (hist.length+1) → g.fst_legal hist act)
            ∧ (Turn_snd (hist.length+1) → g.snd_legal hist act)) →
                g.snd_win_states (act :: hist))
```

We say that a game has a *coherent end*, if:

- we can't satisfy the winning predicates for both players simultaneously.
- If the winning predicate is satisfied for one of the players at a certain history, it stays satisfied for all further legal moves (which we consider dummy moves).

Again, we'll see this requirement in a concrete game when studying Chomp.

The proof of Zermelo's theorem build on two key notions : Conway induction and what we call *staging*. We first discuss Conway induction, which states:

```
lemma Game_World.ConwayInduction [DecidableEq β] (g : Game_World α β)
  (hgw : g.isWL) (hgp : g.playable) (hgn : g.coherent_end)
  (motive : (h : List β) → (g.hist_legal h) → Sort _)
  (hist : List β) (leg : g.hist_legal hist)
  (step : ∀ (h : List β), (leg' : g.hist_legal h) → (∀ (y : List β) (rel :
    R g y h), motive y rel.leg) → motive h leg') :
  motive hist leg :=
```

It makes use of the following relation:

```
structure Rdef (g : Game_World α β) (h H : List β) : Prop where
  extend : ∃ a, H = a :: h
  neutral : g.hist_neutral H
  leg : g.hist_legal H
```

Two histories are in relation if one extends the other by a move, and the histories are legal and neutral (no winning predicate is satisfied). This is the equivalent of `IsOption` relation from Mathlib, and it is indeed well-founded under assumptions of termination, playability and coherent end, as we show in `wfR`.

Before discussing the well-foundedness for a bit, we'll spell out what `Game_World.ConwayInduction` says. For a motive (think of a property, such as "having a winning strategy for one of the players") and a given history, the fact that it is true for all history that extend it legally and neutrally (ie. are in relation `R` with the history) implies that it is true for the current history, then the motive holds for all histories.

Our proof of well-foundedness of `R` is quite cumbersome and it would be interesting to try to shorten it. The approach we took consisted of interpreting well-foundedness via:

```
noncomputable
def Y (r : α → α → Prop) (x : α) (h : ¬ Acc r x) : Nat → {y : α // ¬ Acc
    r y}
| 0 => ⟨x,h⟩
| n+1 =>
    let yn := (Y r x h n).val
    have N : ∃ next, r next yn ∧ (¬ Acc r next) := by
      have ynp := (Y r x h n).prop
      contrapose! ynp
      exact Acc.intro yn ynp
    ⟨Classical.choose N, (Classical.choose_spec N).2⟩

lemma not_Acc (r : α → α → Prop) (x : α) (h : ¬ Acc r x) :
  ∃ Y : Nat → α, (Y 0 = x) ∧ (∀ n : Nat, r (Y (n+1)) (Y n))
```

It state that a relation if a relation isn't well-founded, then we may find a (infinite) sequence `Y` of elements all in relation with the successor term. The next step towards well-foundedness is to translate between the the successive lists that constitute the histories of the game, and a function mapping turns to moves:

```
def Hist_from_moves (moves : ℕ → β) : ℕ → List β :=
    fun t => ((List.range t).reverse.map moves)

def Game_World.moves_from_strats (g : Game_World α β)
  (f_strat : g.fStrategy) (s_strat : g.sStrategy) :
  ℕ → β :=
  fun t =>
    let H := (g.hist_on_turn f_strat s_strat t)
    if T : Turn_fst (t+1) then (f_strat H.val (by rw [H.property.2] ; exact
    T) H.property.1).val else (s_strat H.val (by rw [Turn_snd_iff_not_fst,
    H.property.2] ; exact T) H.property.1).val
```

This allows us to reformulate termination to:

```
def Game_World.isWL_alt (g : Game_World α β) : Prop :=
  ∀ moves : ℕ → β, (∀ t, g.hist_legal (Hist_from_moves moves t)) →
    ∃ T, (g.fst_win_states (Hist_from_moves moves T)) ∨ (g.snd_win_states
    (Hist_from_moves moves T))

lemma Game_World.isWL_iff_isWL_alt [DecidableEq β] (g : Game_World α β)
  (hg : g.playable) :
  g.isWL ↔ g.isWL_alt
```

Essentially, for any function mapping turns to moves of a game that terminates, we may find a turn on which the list made up of the images to that turn satisfies one of the winning predicates, hence "terminating" the game.

Now, if the relation we described wasn't well-founded, then there would be an infinite sequence of histories extending each other, each of which is legal and neutral. If we consider the functions mapping terms to moves for these histories, then we got ourselves a function that violates termination, as described above. So for game that terminate, the relation must be well-founded. Finally, Conway induction amounts to well-founded recursion for that relation.

We now discuss `staging`, the second important aspect to Zermelo's proof. It's defined as:

```
def Game_World.fStrat_staged (g : Game_World α β)
  (f_strat : g.fStrategy) (hist : List β) (leg : g.hist_legal hist) : Prop :=
  ∀ t, (ht : t < hist.length) → (T : Turn_fst (t+1)) →
    f_strat (hist.rtake t) (by rw [List.length_rtake (le_of_lt ht)] ; exact
    T) (g.hist_legal_rtake _ _ leg)
    = ⟨ hist.rget ⟨t,ht⟩ , (g.hist_legal_rtake_fst _ _ T ht leg)⟩

def Game_World.fStrat_wHist (g : Game_World α β) (hist : List β) (leg :
    g.hist_legal hist) :=
  { f_strat : g.fStrategy // g.fStrat_staged f_strat hist leg}
```

A strategy is *staged* wrt. a certain legal history, if for all turns before that history's end, the strategy, when queried at that history up to the turn, returns the entry of that history at that turn, as move to play.

We may then define notions of winning strategies for the staged context:

```
def Game_World.is_fst_staged_win  (g : Game_World α β)
  (hist : List β) (leg : g.hist_legal hist) : Prop :=
  ∃ ws : g.fStrat_wHist hist leg, ∀ snd_s : g.sStrat_wHist hist leg,
  ({g with fst_strat := ws.val, snd_strat := snd_s.val} : Game α β).fst_win

def Game_World.is_snd_staged_win  (g : Game_World α β)
  (hist : List β) (leg : g.hist_legal hist) : Prop :=
  ∃ ws : g.sStrat_wHist hist leg, ∀ fst_s : g.fStrat_wHist hist leg,
  ({g with fst_strat := fst_s.val, snd_strat := ws.val} : Game α β).snd_win

inductive Game_World.has_staged_WL (g : Game_World α β)
  (hist : List β) (leg : g.hist_legal hist) : Prop where
  | wf (h : g.is_fst_staged_win hist leg)
  | ws (h : g.is_snd_staged_win hist leg)
```

Which we may relate to the non-stage context via:

```
lemma Game_World.has_WL_iff_has_staged_WL_empty (g : Game_World α β) :
  g.has_WL ↔ g.has_staged_WL [] Game_World.hist_legal.nil :=
```

Staging is necessary in the proof of Zermelo's theorem for the following technical reason we'll simply try to motivate here. Say, we want to conclude that the player has a winning strategy at a certain history, by using the fact that they can make a move from which on a winning strategy exists from them. The strategy would consist of playing that move, then playing according to this strategy. Recall that our strategies are functions mapping histories to moves. It may now happen that the winning strategy that the player obtains "inductively" after the move actually suggests playing differently, when queried at an earlier stage in the history, as to what's played in the history. We could naively try to enforce playing according to the history up until it is reached, and only then by the winning strategy. But in that context, what do we do when the other player doesn't follow the history ?
Staged strategies, however, allow us to prove the step in Conway induction. Suppose again that the current player has a move that leads them to a history for which they have a *staged* winning strategy (ie. it wins, if both players would stage the game up to that point, playing moves fixed by that history), then playing the move, and then according to the winning strategy *is* a staged winning strategy for the *current* history.

We will now discuss related work.
The blog post [13] does defined Conway induction:

```
def game_ind' (P : game → Prop)
    (H : ∀ l r, P l →  P r → P (Game l r)) :
        ∀ g : game, P g
```

The proof essentially amounts to the recursor of the `game` type. Seeing as it doesn't defined the notions of winning strategies, it doesn't define and prove Zermelo.

Mathlib's formalism [10] similarly proves Conway induction in the form of `PGame.recOn` and `SetTheory.PGame.moveRecOn`. We did not find a theorem equivalent to Zermelo though.

Finally, we discuss Joosten's version [11]. The theorem is proven for "closed" games (`closed_GSgame`), which includes finite games (games where the length of the winning histories are finite), in the final form of `every-position-has-winning-strategy`.
TODO : elaborate ; maybe ask Joosten ? or learn Isabelle

Strategy stealing and Chomp

We now discuss the notion of strategy stealing and illustrate it on the game of "Chomp". To the best of our knowledge, a formalization of these topics has no comparable related work.

Strategy stealing is not a formalized concept in itself, but rather refers to a proof technique/argument to prove that certain games have winning strategies. This is the case for the games of Chomp and the more commonly known game Hex (both in Chapter 1 of [7]). We'll illustrate strategy stealing on the game of Chomp.

Chomp is played on a rectangular gird of tiles as a board, where tiles can be associated with pairs of natural numbers (x, y). Players make take turns selecting a tile of the board, after which all lexicographically dominates tiles get "chomped" off. The last player to select a tile loses the game. We model the game with the following formal definitions:

```
def domi (p q : ℕ × ℕ) : Prop := p.1 ≤ q.1 ∧ p.2 ≤ q.2

def nondomi (p q : ℕ × ℕ) : Prop := ¬ domi p q

def Chomp_state (ini : Finset (ℕ × ℕ)) (hist : List (ℕ × ℕ)) :=
  ini.filter (fun p => ∀ q ∈ hist, nondomi q p)

structure Chomp_law (ini : Finset (ℕ × ℕ)) (hist : List (ℕ × ℕ)) (act : ℕ
    × ℕ) : Prop where
  act_mem : act ∈ ini
  nd : ∀ q ∈ hist, nondomi q act

structure Chomp_win_final (ini : Finset (ℕ × ℕ)) (hist final_h : List (ℕ ×
    ℕ)) (final_a : ℕ × ℕ) : Prop where
  N : Chomp_state ini final_h ≠ ∅
  F : Chomp_state ini (final_a :: final_h) = ∅
  ref : (final_a :: final_h) <:+ hist
```

```
def Chomp_win_fst (ini : Finset (ℕ × ℕ)) (hist : List (ℕ × ℕ)) : Prop :=
  ∃ final_h : List (ℕ × ℕ), ∃ final_a : (ℕ × ℕ), Turn_snd (final_h.length +
     1) ∧ Chomp_win_final ini hist final_h final_a

def Chomp_win_snd (ini : Finset (ℕ × ℕ)) (hist : List (ℕ × ℕ)) : Prop :=
  ∃ final_h : List (ℕ × ℕ), ∃ final_a : (ℕ × ℕ), Turn_fst (final_h.length +
     1) ∧ Chomp_win_final ini hist final_h final_a

def preChomp (height length : ℕ) : Symm_Game_World (Finset (ℕ × ℕ)) (ℕ × ℕ
    ) where
  init_game_state := Chomp_init height length
  fst_win_states := fun hist => Chomp_win_fst (Chomp_init height length) hist
  snd_win_states := fun hist => Chomp_win_snd (Chomp_init height length) hist
  transition := fun hist act => Chomp_state (Chomp_init height length) (act
    :: hist)
  law := fun hist act =>  if Chomp_state (Chomp_init height length) hist ≠ ∅
                          then Chomp_law (Chomp_init height length) hist act
                          else True
```

The board of the game staring from board `ini` and playing moves `hist` is given by `Chomp_state`, wich filters the tiles fro the initial board, keeping those that are not dominated by any of the previously played tiles. `Chomp_law` describes what it means for an actions to be legal at a stage in the game: it should be a tile in the initial board, and it shouldn't be dominated by any of the previous moves. We note that in the actual definition of the game, we let the law apply only if the state is non-empty, as for an empty board, we consider the game to be over. This is required for the game to be `playable` in the sense defined in the previous section. Indeed, for there to always be legal moves to play, the law above fails to yield this property when the board is empty, as tile $(0, 0)$ must have been selected, which dominates all tiles, so that none are legal to play anymore. Finally, the winning condition is achieved when the history of moves has a suffix ($\approx$ a prior history) where the state was non-empty, and became empty on the next turn, and this turn was that of the opposite player (recall that one *loses* by taking the last tile). The reason we require the actual winning condition to be satisfied by a suffix is so as to satisfy the notion of `coherent_end` we defined in the previous section. Indeed, all histories extending a winning one will also be winning, as desired.

We will not discuss the proofs that Chomp satisfies the conditions of Zermelo's theorem, and skip ahead to a discussion of strategy stealing. The argument for Chomp goes as follows. Assume for contradiction that the second player had a winning strategy. The first player can steal it as follows: they play the topmost-outermost tile, and then play according to the second players winning strategy, in a fake game where the actual second player would go first. The reason this works is that the board, in all turns $t \geqslant 2$ in the actual game can be translated to the turns t in the fake game, so that the first player winning as the second in the fake game corresponds to them winning the true game.

To make this argument formal, we will define the moves that allow to translate the real game to a fake one:

```
structure Bait (g : zSymm_Game_World α β) (trap : β) : Prop where
  leg_fst : g.law [] trap
  leg_imp : ∀ hist, ∀ act, g.hist_legal (hist) → g.law (hist ++ [trap]) act
    → g.law (hist) act
  leg_imp' : ∀ hist, ∀ act, g.hist_legal (hist) → (Z : ¬ hist = []) →
    hist.getLast Z = trap → Turn_fst (hist.length + 1) → g.law
    hist.dropLast act → g.law hist act

structure Stealing_condition (g : zSymm_Game_World α β) where
  trap : β
  hb : Bait g trap
  fst_not_win : ¬ g.snd_win_states []
  wb : ∀ hist, g.hist_legal hist → g.snd_win_states (hist ++ [trap]) →
    g.fst_win_states hist
```

We call a move a *bait* if:

- it is a legal first move
- has the property that for all histories, the next moves are legal when they were in a history that was preceded by the move
- has the property that for all histories that have ‘trap‘ as first move, the next moves are legal if they are wrt. these histories, with the first move skipped.

In Chomp, the topmost-outermost tile is the bait, and we show this in `Chomp_Bait`.
Next, a game satisfies the *stealing conditions* if:

- it has a `trap` move that serves as bait
- the initial state is not a winning state of the second player
- histories that start with `trap` and yield second wins yield first wins if `trap` is skipped

We show that Chomp satisfies these conditions in `Chomp_stealing_condition`.
Finally, we may define the stolen strategy for the first player, and the fake strategy that mimics that of the actual second player, in the fake game where it would go first, in the following, respectively:

```
noncomputable
def stolen_strat (g : zSymm_Game_World α β)
  (trap : β) (hb : Bait g trap) (ws : g.sStrategy) : g.fStrategy :=
  -- assumes ws is the supposed winning strategy of the second player we want
    to steal
  fun h ht hl =>  if M : g.hist_legal (h ++ [trap])
                  then  let move := (ws (h ++ [trap]) (sorry) M)
                        -- given a history, we consider the response move of
    the winning strat in a ficticious
                        -- history where the trap was played, and play it.
                        ⟨move.val, hb.leg_imp h move.val hl (move.prop)⟩
                  else g.toGame_World.exStrat_fst g.hgp h ht hl
                  -- we expect the condition to be true, so we spam default
    moves here

noncomputable
def pre_stolen_strat (g : zSymm_Game_World α β)
  (trap : β) (hb : Bait g trap) (s_strat : g.sStrategy) : g.fStrategy :=
  -- s_strat is expected to play against the stolen strategy ;
  -- This strategy mimics it as a ‘fStrategy‘, playing the ‘trap‘ as first
    move, then copying s_strat
  fun h ht hl =>  if Z : h = []
                  then  ⟨trap, (by rw [Z] ; exact hb.leg_fst)⟩
                        -- initially, play ‘trap‘
                  else  if M : h.getLast Z = trap
                        then  let move := s_strat h.dropLast (sorry) (sorry))
                              -- play the response of s_strat in a history
    that ignores ‘trap‘
                              ⟨move.val, hb.leg_imp' h move.val hl Z M ht
    move.prop⟩
                        else  g.toGame_World.exStrat_fst g.hgp h ht hl
```

We omitted proofs, but left in explanatory comments in the above code. The correspondence between the true and the fake game are given in `History_of_stealing`. Now, by Zermelo's theorem, if the first player didn't have a winning strategy for Chomp in the first place, then the second player must have one. However, if the second player had a winning strategy, the the first player could steal it, and win the fake game with it, which corresponds to winning the actual game. Hence, the first player has a winning strategy for Chomp, and we show this via `Strategy_stealing`, in `Chomp_is_fst_win`.

Positional games, paring strategies, Tic-tac-toe

Finally, we discuss the notions of positional games and pairing strategies for them, and illustrate these notions on the game of Tic-tac-toe. The main informal source for these topics is [8].

Positional games refer to a class of games played by claiming, or "coloring", the tiles of a board. Once a tile is colored by a player, the other may not color it. There are sets of tiles of the board, referred to as winning sets, that yield the following winning condition: to win, a player must have colored all tiles of a winning set. The game ends in a draw if all tiles are colored, yet no winning set is colored. A typical example for a positional game is Tic-tac-toe, where the board is a 3 by 3 grid, and the winning sets correspond to the lines and diagonals in the grid. Though we won't reproduce the full definition of a positional game here, we show the main components:

```
def PosGame_trans [DecidableEq α] (hist : List α) : α → Fin 3 :=
  fun p => if  p ∈ hist
           then if Turn_fst ((hist.reverse.indexOf p) + 1)
                then 1
                else 2
           else 0


def PosGame_win_fst [DecidableEq α] [Fintype α] (win_sets : Finset (Finset α
    )) (ini : α → Fin 3) (hist : List α) : Prop :=
  ∃ w ∈ win_sets, w ⊆ Finset.filter (fun x => (State_from_history ini (fun
    hist act => PosGame_trans (act :: hist)) (fun hist act => PosGame_trans
    (act :: hist)) hist) x = 1) Finset.univ


def PosGame_win_snd [DecidableEq α] [Fintype α] (win_sets : Finset (Finset α
    )) (ini : α → Fin 3) (hist : List α) : Prop :=
  ∃ w ∈ win_sets, w ⊆ Finset.filter (fun x => (State_from_history ini (fun
    hist act => PosGame_trans (act :: hist)) (fun hist act => PosGame_trans
    (act :: hist)) hist) x = 2) Finset.univ
```

We model the board by a finite type α. The state of the board will be encoded as a function from α to $\{0, 1, 2\}$, mapping a tile to its color, where 0 represents an uncolored tile, 1 a tile colored by the first player and 2 a tile colored by the second. To determine that color of a tile at a stage in the game, we look if the tile was played by on of the players: if not, it is uncolored, and if it is, the index of it in the history indicates the turn it was played, on and hence the player who played it, coloring it by their color. In this context, the winning conditions are that there is a winning set which is contained in the preimage under the state map of the color of the corresponding player. Playing a tile will be legal in a positional game if it hasn't been played yet, unless the game is over, in which case any move, which is considered a dummy move, is legal. A draw is attained if no winning sets are monochromatic and no tile is uncolored.

Before discussing pairing strategies, we which to discuss Tic-tac-toe, which in turn requires discussing combinatorial lines. Indeed, we'll play Tic-tac-toe in dimension D, on a cubical grid of side-length n. The winning sets will be the combinatorial lines of the cube. A combinatorial lines is a set of grid-points who's coordinates form either the sequences $0, 1, 2, ..., n-1$ or $n-1, n-2, ..., 0$, or a constant sequence, and where not all such coordinate sequences are constants (otherwise, it would be a single point). We express them as:

```
variable (D n : ℕ)
variable (hn : n ≠ 0)

def Opp (k : Fin n) : Fin n :=
  ⟨n - 1 - k.val, sorry⟩

inductive in_de_const (s : Fin n → Fin n) : Prop
| cst : (∃ x : Fin n, ∀ i : Fin n, s i = x) → in_de_const s
| inc : (∀ i : Fin n, s i = i) → in_de_const s
| dec : (∀ i : Fin n, s i = Opp n hn i) → in_de_const s

structure seq_is_line (s : Fin n → (Fin D → Fin n)) : Prop where
  idc : ∀ d : Fin D, in_de_const n hn (fun j : Fin n => (s j) d)
  non_pt : ¬ (∀ d : Fin D, (∃ x : Fin n, ∀ j : Fin n, s j d = x))

structure seq_rep_line (s : Fin n → (Fin D → Fin n)) (l : Finset (Fin D →
    Fin n)) : Prop where
  line : seq_is_line D n hn s
  rep : l = Finset.image s .univ

def is_combi_line (can : Finset (Fin D → Fin n)) : Prop :=
  ∃ s : Fin n → (Fin D → Fin n), seq_rep_line D n hn s can
```

Here `Fin D → Fin n` is though of as a point in $\{0, 1, .., n-1\}^D$ and `Fin n → (Fin D → Fin n)` is a sequence of n such points. With them, we may define the game of Tic-tac-toe:

```
variable (D n : ℕ)
variable (hn : 1 < n)

open Classical

noncomputable
def TTT_win_sets : Finset (Finset (Fin D → Fin n)) := Finset.univ.filter
    (is_combi_line D n (strengthen n hn))

noncomputable
def TTT := Positional_Game_World (TTT_win_sets D n hn)
```

Now, we shall informally discuss the notion of pairing strategies and how they apply to Tic-tac-toe. If for each winning set in a positional game, we may find a pair of tiles, such that all tiles are different from one another (within and among pairs), then the following strategy may be played: if the adversary plays one of the tiles from a pair, play the other next, or play random moves if the other tile has already been claimed. We can show that for such a strategy, no winning set will ever be monochromatic: if it were, consider the first time one of the tiles was colored by the winning player, so that the other must not have been, or was already colored by the adversary, then with this strategy, the other tile will be of the adversaries color at the end of the turn, so that the winning set can in fact not have been monochromatic.

The question now is if such pairings exist for Tic-tac-toe. It tuns out that if $n \geqslant 3^D - 1$, there is. To seen this, we begin by considering the following matching problem : consider a bipartite graph with the points of $\{0, 1, .., n-1\}^D$ on one side, and two points for each combinatorial line on the other side. We connect vertices by an edge if the points are incident (are in) the line. Now, if we can find a matching of the lines-bipartition-set into the point-bipartition-set, we may use these points for a pairing, as we'll have exactly two points for each combinatorial line (which was represented twice in the bipartition-set), and all points are different. Hall's theorem, who's formalization has made the object of the paper [16] by Alena Gusakov, Bhavik Mehta, and Kyle Miller, provides a condition for such a matching to exist: for each subset of line-vertices, the total number of neighbors must be larger. Using some simple double-counting, we may derive a relation between n and D for which this is true.

For each combinatorial line passing through a given point, the coordinate sequences are of three types: increasing, deceasing, or constant with value that point's coordinate. There are at most 3^D such combinations of coordinate sequences. We discard the one in which all sequences are constant, so that we can upper-bound the number of combinatorial lines by $3^D - 1$. Since one line was accounted for twice in this manner (inverting all increasing and decreasing sequences yields the same set of points), we may actually use $\dfrac{3^d - 1}{2}$ as upper-bound.
Back to Hall's application in our context. To compare the size of line-vertex subset S to its neighbourhood $N(S)$ of points in those lines, we'll estimate the edges between them in two ways. First, since each line contains n points, we know that $\displaystyle\sum_{L \in S} |\delta(L)| = |S|n$.
On the other bipartition set, these edges are counted in $\displaystyle\sum_{p \in N(S)} |\delta(p)|$, which can be bounded by recalling that any point is in at most $\dfrac{3^d - 1}{2}$ lines, and each line is represented twice in the line-bipartition,so that $\displaystyle\sum_{p \in N(S)} |\delta(p)| \leqslant 2|N(S)|\frac{3^d - 1}{2}$. Combining the bounds, we get $|S|\dfrac{n}{3^d - 1} \leqslant |N(S)|$. So in the case that $n \geqslant 3^d - 1$, we get Hall's condition, and a pairing strategy exists.

Formally, we express the notion of a pairing for a pairing strategy via:

```
structure pairProp {win_sets : Finset (Finset α)} (win_set : win_sets) (p : α
    × α) : Prop where
  dif : p.1 ≠ p.2
  mem_fst : p.1 ∈ win_set.val
  mem_snd : p.2 ∈ win_set.val

structure pairDif (a b : α × α) : Prop where
  strait_fst : a.1 ≠ b.1
  strait_snd : a.2 ≠ b.2
  cross_fst : a.1 ≠ b.2
  cross_snd : a.2 ≠ b.1

structure Pairing_condition [DecidableEq α] [Fintype α] (win_sets : Finset
    (Finset α)) (pairing : win_sets → (α × α)) where
  has_pairing : ∀ w : win_sets, pairProp w (pairing w)
  pairing_dif : ∀ w v : win_sets, w ≠ v → pairDif (pairing w) (pairing v)
```

The pairing is a function mapping winning sets to pairs of tiles, such that the tiles belong to the winning set and all tiles are different from one another. Now, the pairing strategy, regardless of the player, may be expressed as follows:

```
noncomputable
def Pairing_StratCore [Inhabited α] [DecidableEq α] [Fintype α] (win_sets :
    Finset (Finset α)) (pairing : win_sets → (α × α)) :
  (hist : List α) → (leg : (Positional_Game_World win_sets).hist_legal hist)
    → α :=
  fun hist leg =>
    let spam :=
      if T : Turn_fst (hist.length + 1)
            then
              Classical.choose (((Positional_Game_World_playable win_sets)
    hist (leg)).1 T)
            else
              Classical.choose (((Positional_Game_World_playable win_sets)
    hist (leg)).2 (by rw [Turn_snd_iff_not_fst] ; exact T))
    match hist with
    | last :: _ =>
        if hxf : ∃ w : win_sets, last = (pairing w).1 -- if the last move
    colored the first tile of the pair of a winning sets ...
        then
          let other := (pairing (Classical.choose hxf)).2
```

```
        if other ∈ hist
        then spam
        else other -- play the other tile, or spam if it's already colored
      else
        if hxs : ∃ w : win_sets, last = (pairing w).2 -- symmetric case of ↑
        then
          let other := (pairing (Classical.choose hxs)).1
          if other ∈ hist
          then spam
          else other
        else
          spam
  | [] => spam -- if we're first to move, spam
```

If the adversary played on tile of the pair, this strategy plays the other. If not, playability of positional game ensures we may spam arbitrary (but unspecified) legal moves. With a lot of effort, we may prove:

```
theorem Pairing_Strategy [Inhabited α] [DecidableEq α] [Fintype α] {win_sets
    : Finset (Finset α)} {pairing : win_sets → (α × α)}
  (win_sets_nontrivial : ∅ ∉ win_sets) (hg : Pairing_condition win_sets
    pairing) :
  (Positional_Game_World win_sets).is_draw_at_worst_snd
```

Finally, we simply list the results on combinatorial lines, as a road-map to the code:

```
private def the_inj (c : (Fin n → Fin D → Fin n)) : Fin D → Fin 3 :=
  fun d =>
    if (∃ x : Fin n, ∀ i : Fin n, c i d = x)
    then
      0
    else
      if (∀ i : Fin n, c i d = i)
      then
        1
      else
        2


lemma incidence_ub (p : Fin D → Fin n) :
  (Finset.univ.filter (fun c : Finset (Fin D → Fin n) => is_combi_line D n
    (strengthen n Hn) c ∧ p ∈ c)).card ≤ (3^D - 1)/2 :=
```

To show that at most $3^D - 1$ combinatorial lines may pass through a given point, we inject the type of sequences representing lines into `Fin D → Fin 3` (or $\{0,1,2\}^D$, informally), with the map `the_inj`. The main result is then `incidence_ub`, as shown above.

We simply mention that `combi_line_repr_card` is the lemma used to show that two exactly two sequences of points represent a combinatorial line. The key steps that follow are:

```
def line_set_neighbours (l : {c : Finset (Fin D → Fin n) // is_combi_line D
    n (strengthen n hn) c} × Bool): Finset (Fin D → Fin n) :=
  Finset.univ.filter (fun p => p ∈ l.1.val)

lemma Hall_condition (ls : Finset ({c : Finset (Fin D → Fin n) //
    is_combi_line D n (strengthen n hn) c} × Bool))
  (h : n ≥ 3^D - 1) :
  ls.card ≤ (Finset.biUnion ls (line_set_neighbours D n hn)).card
```

Since we wish to represent lines twice in the bipartite graph we wish to use Hall's theorem on, we consider pairs of a line and a boolean, though any type of size 2 would work just as well. Piecing things together, we get:

```
theorem TTT_is_draw_at_worst_snd : (TTT D n hn).is_draw_at_worst_snd
```

Further research

The formalism we developed is in no way an accomplished one. We are currently experimenting with restricting strategies to answer only to histories of moves in which the game hasn't ended:

```
structure Game_World.hist_neutral (g : Game_World α β) (hist : List β) :
    Prop where
  not_fst : ¬ g.fst_win_states hist
  not_snd : ¬ g.snd_win_states hist

def Game_World.fStrategy (g : Game_World α β) :=
  (hist : List β) → (Turn_fst (hist.length+1)) →
  (g.hist_legal hist) → (g.hist_neutral hist) →
  { act : β // g.fst_legal hist act}

def Game_World.sStrategy (g : Game_World α β) :=
  (hist : List β) → (Turn_snd (hist.length+1)) →
  (g.hist_legal hist) → (g.hist_neutral hist) →
  { act : β // g.snd_legal hist act}
```

Much of the API we developed may be extended. For instance, we didn't develop the class of symmetric games that allow for draw states, and more fundamentally, we haven't proven Zermelo's theorem for games with draw states.

Finally, we would like to attempt to make our results computable. For example, though we make use of classical choice for pairing strategies, we could instead be specific. In Tic-tac-toe, the `spam` from the pairing strategies could consist of choosing the lexicographically smallest tile not in the current history, or the dummy tiles 0 if all tiles are colored. Then, the pairing strategies could be `#eval`uated, as with those from Pick-up-bricks.

References

[1] Moura, Leonardo de, and Sebastian Ullrich. "The Lean 4 theorem prover and programming language." Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28. Springer International Publishing, 2021.

[2] Bove, Ana, Peter Dybjer, and Ulf Norell. "A brief overview of Agda–a functional language with dependent types." Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22. Springer Berlin Heidelberg, 2009.

[3] Barras, Bruno, et al. The Coq proof assistant reference manual: Version 6.1. Diss. Inria, 1997.

[4] Carneiro, Mario. "Lean4Lean: Towards a formalized metatheory for the Lean theorem prover." arXiv preprint arXiv:2403.14064 (2024).

[5] The mathlib community, The Lean mathematical library. Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020

[6] https://leanprover-community.github.io/papers.html

[7] Kent, Deborah, and Matt De Vos. Game theory: a playful introduction. American Mathematical Society, 2017.

[8] Hefetz, D., Krivelevich, M., Stojaković, M., Szabó, T. (2014). Positional games (Vol. 44). Basel: Birkhäuser.

[9] Longbottom, Isabel. "Combinatorial Game Theory in Lean." (2019). https://isabel-prime.github.io/files/combinatorial-game-theory-in-lean.pdf

[10] https://github.com/leanprover-community/mathlib4/tree/master/Mathlib/SetTheory/Game

[11] Joosten, Sebastiaan JC. "Gale-Stewart Games." (2024). https://www.isa-afp.org/entries/GaleStewart_Games.html

[12] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS. Springer, 2002.

[13] https://poleiro.info/posts/2013-09-08-an-introduction-to-combinatorial-game-theory.html

[14] https://github.com/sven-manthe/A-formalization-of-Borel-determinacy-in-Lean

[15] Dittmann, Christoph. "Positional determinacy of parity games.")https://www.isa-afp.org/entries/Parity_Game.html

[16] Gusakov, Alena, Bhavik Mehta, and Kyle A. Miller. "Formalizing Hall's Marriage Theorem in Lean." arXiv preprint arXiv:2101.00127 (2021).

Address

Email address: `jaeckle@zib.de`