A Formalization of two player turn based Games

Yves Jäckle
TU Berlin, Germany

Abstract

We give formal foundations and verify theorems about two player turn based finite games in Lean. Games are modeled via predicates on lists of possible previous actions taken by the players, deciding which further moves are allowed and whether the game has reached a winning or drawing stage. The player's strategies are modeled as functions mapping valid lists of possible previous actions to the next action taken by the player. In this context, we formalize the notion of a winning strategies, the notion of strategy stealing, which we exemplify with the game of "Chomp", and finally the notion of pairing strategies for positional games, which we exemplify with the game of d-dimensional Tic-Tac-Toe.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Interactive theorem proving, Formalization, Lean, Game theory, Combinatorial games, Positional games

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding (Optional) general funding statement ...
Yves Jäckle: [funding]

Acknowledgements I want to thank Sven Manthe and Violeta Hernández Palacios for insights on the in-/formal aspects of Gale-Stewart games and combinatorial games, respectively.

1 Introduction

Game theory

Game Theory is a broad subject and research area. Our interests lie in the theory of two player turn based games, such as Hex [8], Nim [3], or Shannon's switching game [14] to name the perhaps most commonly known. In such kind of games, two players take turns changing a state - the board of the game - by playing one of a selection of (usually finitely many) available actions, until (assuming the game ends) the game is declared a win for one of the players, or declared a draw. For example, we'll give a brief description of the game of Pick-Up-Bricks, which we'll formalize in a later section: two players take turns picking either one or two bricks from a stack of bricks, and the last player to take a brick wins the game. We'll see that the player that goes second has a "winning strategy" if the initial stack of bricks is divisible by three, i.e. a way of always winning the game, no matter how the opponent plays.

The formal foundations of this part of Game Theory have been the subject of long and fruitful research. Indeed, there are multiple ways to develop a formalism studying games at an abstract level, with which to represent such concrete games, and in each such way, there is quite some work to be done to make sense of notions such as "winning strategies", that are used often and rather loosely in the study of concrete games. The interested reader may consult [20] for example, to find an account of the beginnings of formal Game Theory, via a study of the history of Zermelo's theorem [21]. The two most prominent and accomplished formal foundations of this part of Game Theory are Gale-Stewart games [9] and Conway's combinatorial games [5], both of which have made the object of formalization in interactive theorem provers, as we'll soon discuss. The intension of our work was to design a formalism closest to Gale-Stewart games, that lends itself smoothly for the study of concrete games,

and exemplify its use on the study of the games of Pick-Up-Bricks [13], Chomp [7], and Tic-Tac-Toe [11], with the assistance of the Lean interactive theorem prover.

Formalization

Lean [19] is an ITP (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 [4] or Coq [1]. Lean hosts Mathlib [18], an ever-growing library providing unified formal foundations to many fields of mathematics.

Contributions

We define the notions of games - in particular positional games [11] -, of strategies and of winning strategies. With these notions, we prove Zermelo's theorem, we give conditions to an arguments know as "strategy stealing" and "pairing strategies". We apply these results to the study of the games of Pick-Up-Bricks [13], Chomp [7], and Tic-Tac-Toe [11].
Our code is publicly available at URL `https://github.com/Happyves/LeanGamesFix` and builds on Lean v.4.5.

Related Work

Combinatorial game theory was already given formal foundations in Lean in work initiated by Isabel Longbottom [15] in 2019. Currently, Mathlib contains formal foundations of combinatorial game theory (as well as applications to the games of Nim, Domineering and Poset games), with main contributors being: Reid Barton, Mario Carneiro, Markus Himmel, Isabel Longbottom, Kim Morrison, Apurva Nakade, Violeta Hernández Palacios and Fox Thomson. In Isabelle/HOL [12], foundations of Gale-Stewart games were given by Sebastiaan J. C. Joosten [12] in 2021, based on the work of Christoph Dittmann [6]. Recently, Sven Manthe has implemented Gale-Stewart games and Borel determinacy in Lean. We are not aware further published work on formal combinatorial game theory.

Overview

First, in section 2.1, we discuss our formalism. To do so, we present the game of Pick-Up-Bricks [13] that will serve as running example to illustrate the formalism on a concrete game. We define the notions of a game, of the history of previously played actions, and of strategies. Next, we define the notion of a terminating game and give meaning to the notion of a "winning strategy". Then, in section 2.2, we briefly discuss the related formalisms of Gale-Stewart games and of combinatorial games. In section 3.1, we discuss our implementation of Zermelo's theorem. We then make use of this key result in section 3.2, in which we study the game of Chomp [7], which is shown to have a winning strategy for the second player, via an argument known as "strategy stealing". The last use case of our formalism is given in section 3.3 in which we study the positional game of d-dimensional Tic-Tac-Toe [11], showing via an argument known as "pairing strategies" that under a condition on dimension an gird size, the players have "drawing strategies". Finally, in our concluding section 4, we discuss future work surrounding the formalization of game theory.

2 Formal Game theory

2.1 Our Formalism

Pick-Up-Bricks

First we introduce the game of Pick-Up-Bricks [13], which will serve as running illustrative example for our formalism. We begin by giving the relevant informal definitions:

▶ **Definition 1** (Pick-Up-Bricks). *Pick-Up-Bricks is played by two players taking turns picking either* 1 *or* 2 *bricks from the top of a stack of* n *bricks. The last player to pick a brick wins the game.*

▶ **Definition 2** (Strategies). *Given a game, a strategy for a player is a decision procedure that is queried by the player at the current state of the game, to determine the next move to be played, if doing so is still possible.*

▶ **Definition 3** (Winning strategies). *Given a game, a winning strategy for a player is a strategy such that, for all strategies of the opponent, the game is won by the player.*

Clearly, the game of Pick-Up-Bricks always terminates, in the sense that the winning condition is reached in a finite number of turns, since the number of bricks of the finite stack strictly decreases at each turn. It also allows for winning strategies:

▶ **Proposition 4** (Winning strategy for Pick-Up-Bricks). *If the initial number of bricks* n *is divisible by* 3*, then the player that goes second has a winning strategy.*

Proof. We claim that the following constitutes a winning strategy: if in the previous turn, the opponent took 1 brick, take 2, and if they took 2 bricks, then take 1. Now, consider any strategy for the first player and consider the game that follows from the players playing according to these strategies. We can show by induction that the stack of bricks is divisible by 3 initially and then precisely after the second player's moves. Indeed, after each round (two turns), $1 + 2 = 2 + 1 = 3$ bricks were taken, so that the stack remains divisible by 3. Hence, since 0 is divisible by 3, it is the second player that must have taken the last brick(s), thereby winning them the game. ◀

The discussion so far was riddled with impreciseness. We will now give formal meaning to the developed notions and results.

Basics

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
```

In the above listing and in those that will follow, we use semicolons to delimit structure fields, though this is invalid syntax in Lean. We shall also omit fields, proofs and definitions to ease exposition. 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 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` indicates that the turn was the first players, and simply reduces to saying that the turn is odd, while `Turn_snd` indicates 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 it's 1 or 2.
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}
```

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 joining the played move and a proof that this move is legal.
The predicate deciding whether a list of moves constitutes a history of legal moves:

```
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)
```

These definitions avoid having to let strategies return dummy values for histories not made up of legal moves. In the current formalism, strategies must however play dummy values for histories in which we consider the game to have terminated, and we'll see the consequences that arise due to this further on in our discussion. 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 also 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
```

Termination

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 ···

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)
```

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
```

In words: the corresponding player has a strategy that beats all possible strategies of the other player. For Pick-Up-Bricks, we can now defined the winning strategy of proposition 4:

```
def pub_win_strat : (PickUpBricks init_bricks).sStrategy :=
  fun hist _ _ =>   match con : hist with
                    | [] => False.elim (by contradiction)
                    | last :: _ => if last = 2 then ⟨1, ···⟩ else ⟨2, ···⟩
```

The "round-invariant" we described is then formalized as:

```
lemma loop_invariant (win_hyp : 3 | init_bricks)
  (f_strat : (PickUpBricks init_bricks).fStrategy) :
    let g : Symm_Game ℕ ℕ := { (PickUpBricks init_bricks) with
    fst_strat := f_strat, snd_strat := pub_win_strat init_bricks} ;
  ∀ turn, Turn_snd turn →
    3 | bricks_from_ini_hist init_bricks (g.hist_on_turn turn)
```

We us it to prove proposition 4:

```
theorem PUB_snd_win (win_hyp : 3 | init_bricks) :
    (PickUpBricks init_bricks).is_snd_win
```

2.2 Related Formalisms

Gale-Stewart Games

Gale-Stewart games have been formalized in Isabelle by Joosten [12] based on work of Dittman [6]. The central notion is similar to our notion of a history of moves, in the form of lists where even indexed moves correspond to those of the player "Even", and the odd indexed ones to those of the player "Odd". Games 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`. Note that the sequences defining winning games must have the same fixed (but possibly infinite) length, and one can represent games terminating earlier by including all sequence having a winning one as prefix in the winning set. 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.
In Lean, Sven Manthe recently formalized Gale-Stewart games and Borel determinacy in [16], following [17], simultaneously and independently of our work. There, games are defined as sets of lists that can be interpreted as the history of moves taken by the players. Such a set is required to be stable under taking list prefixes, so that it may be thought of as a tree, with these lists as vertices, and where the children of a list correspond to those where a legal move was appended. One then considers a set of streams (infinite lists) among the body of the tree (streams all of whose prefixes are in the tree), and considers a game to be won by a designated player if the stream corresponding to the moves played belongs to this set. Strategies are defined as functions taking lists from the tree and returning a list that extends the input by a legal move.

Conway's Combinatorial Games

Finally, we discuss the formalism initially developed by Longbottom [15] and adopted and further developed by Mathlib [18]. 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
```

Two players "Left" and "Right" play moves until a player runs out of available moves, in which case their opponent wins. `PGame` can both be though of as the state of the game and as a tree, where branches are indexed by the terms of a branch type, representing the remaining possible ways the game can proceed. One can encode turn based games by letting one of the action types be a uninhabited type. To develop the notion of winning strategies, the relation `SetTheory.PGame.le` is used. As before, we consider a game to be lost by a player if they have no moves left to play on their turn, in the sense that the corresponding move type in the games constructor is uninhabited. By letting 0 denote an empty, and hence losing-position-game, the theorem below 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. Hence, the notion of a winning strategy is implicitly contained in this relation.

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

3 Aspects of Formal Games

3.1 Zermelo's Theorem

Introduction

We want to formalize the following:

▶ **Theorem 5** (Zermelo's theorem). *Games that terminate and do so in either a win or a loss for the players have a winning strategy for one of the players*

Proof. The proof is by well-founded induction. If in the current state of the game, there is a move such that the current player has a winning strategy from the stage thereon, they may play that move, and then 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). ◀

Zermelo's theorem can be generalized, for example to Borel determinacy [17].

Formalization

Our statement of Zermelo's theorem is the following, where `Game_World.has_WL` states that the first or the second player have a winning strategy.

```
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 discussed (`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. 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 : ···
```

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

```
theorem 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). We prove that it is well-founded under assumptions of termination, playability and a coherent end. To do so, the approach we took consisted of interpreting well-foundedness via:

```
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 states 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 (···) H.property.1).val else
    (s_strat H.val (··· H.property.1).val
```

This allows us to rephrase 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 up to that turn satisfies one of the winning predicates. 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 obtained a function that violates termination, as described above. So for games 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 the proof of Zermelo's theorem.

```
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) (···) (···) = ⟨ hist.rget ⟨t,ht⟩ , ···⟩

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 such a 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

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)
```

The translation to staged strategies is crucial in the proof of the induction step of Zermelo's theorem. Which we may relate to the non-staged 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 :=
```

Finally, we define structures corresponding to games that satisfy the conditions to Zermelo's theorem, such as `zSymm_Game_World`, which we'll use in the next section.

3.2 Strategy Stealing and Chomp

Introduction

Chomp [7] is played on a rectangular gird of tiles as a board, where tiles can be associated with pairs of natural numbers (x, y). Players take turns selecting a tile of the board, after which all tiles dominated by the selected one (those for which both respective coordinates are greater or equal then those of the played) get "chomped" off (ie. may not be played further-on). The last player to select a tile loses the game.

▶ **Proposition 6** (Strategy stealing for Chomp). *The first player has a winning strategy for chomp.*

Proof. Assume for contradiction that this wasn't the case, so that by Zermelo's theorem, the second player had a winning strategy. The first player can steal it as follows: they play the topmost-outermost tile of the rectangle, and then play according to the second players winning strategy, in a rest of the game, where the actual second player would be considered to go first. All turns $t+1$ in the actual game can be translated to the turns t in the "shifted" game, so that the first player winning as the second in the "shifted" game corresponds to them winning the actual game. ◀

Strategy Stealing

To make the above argument formal, we will define the moves that allow to translate the real game to a "shifted" 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 this bait move as first move, the next moves are legal if they are wrt. these histories, with the first move skipped.

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

Then, we may define the stolen strategy for the first player, and the "shifted" strategy that mimics that of the actual second player, in the "shifted" 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 :=
  fun h ht hl =>  if M : g.hist_legal (h ++ [trap])
                  then  let move := (ws (h ++ [trap]) (···) M) ; ⟨move.val, ···⟩
                  else g.toGame_World.exStrat_fst g.hgp h ht hl

noncomputable def pre_stolen_strat (g : zSymm_Game_World α β)
  (trap : β) (hb : Bait g trap) (s_strat : g.sStrategy) : g.fStrategy :=
  fun h ht hl =>  if Z : h = [] then  ⟨trap, ···⟩
                  else  if M : h.getLast Z = trap
                        then  let move := s_strat h.dropLast (···) (···))
                              ⟨move.val, ···⟩
                        else  g.toGame_World.exStrat_fst g.hgp h ht hl
```

We simply mention that the correspondence between the true and the "shifted" game are given in our lemma `History_of_stealing`, and that in our theorem `Strategy_stealing`, we show that if the stealing conditions are satisfied, the first player has a winning strategy.

Chomp

We model the game of Chomp 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_init (height length : ℕ) :=
    (Finset.range (length+1)) × (Finset.range (height+1))

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 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 after playing moves `hist` is given by `Chomp_state`, which filters the tiles from 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 (a prior history) in which 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. In Chomp, the topmost-outermost tile is the bait:

```
lemma Chomp_Bait {height length : ℕ} (h : height ≠ 0 ∨ length ≠ 0) :
    Bait (Chomp height length) (length, height)
```

Using this, we show that Chomp satisfies the stealing conditions in our theorem `Chomp_stealing_condition` and that it therefore has a winning strategy for the first player, in our theorem `Chomp_is_fst_win`, using strategy stealing.

3.3 Pairing Strategies and Tic-Tac-Toe

Introduction

Positional games [11] 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 monochromatic. 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. Now, we shall informally discuss the notion of pairing strategies and how they apply to d-dimensional Tic-tac-toe, which is the positional game played on the points of $\{0, 1, .., n-1\}^D$, where the winning sets are combinatorial lines. If for each winning set of 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 turn on which 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 adversary's color at the end of the next turn, so that the winning set can in fact not have been monochromatic. The question now is whether such pairings exist for Tic-tac-toe.

▶ **Proposition 7** (Pairing strategies for d-dimensional Tic-Tac-Toe). *If $n \geqslant 3^d - 1$, then d-dimensional Tic-Tac-Toe on a cube of side-length n admits pairing strategies.*

Proof. 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 vertices for each combinatorial line on the other side. We connect vertices by an edge if the points are incident to 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 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, as follows. 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 $\frac{3^d-1}{2}$ as upper-bound. In the context of Hall's theorem, in order to compare the size of line-vertex subset S to its neighborhood $N(S)$ 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 $\sum_{L \in S} |\delta(L)| = |S|n$. On the other bipartition set, these edges are counted in $\sum_{p \in N(S)} |\delta(p)|$, which can be bounded by recalling that any point is in at most $\frac{3^d-1}{2}$ lines, and each line is represented twice in the line-bipartition,so that $\sum_{p \in N(S)} |\delta(p)| \leqslant 2|N(S)|\frac{3^d-1}{2}$. Combining the bounds, we get $|S|\frac{n}{3^d-1} \leqslant |N(S)|$. So in the case that $n \geqslant 3^d - 1$, we get Hall's condition, and a pairing strategy exists. ◀

Hall's theorem was formalized by Alena Gusakov, Bhavik Mehta, and Kyle Miller [10].

Positional games

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
```

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 check if the tile was played by one 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.

Pairing Strategies

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 (···))
    match hist with
    | last :: _ => if hxf : ∃ w : win_sets, last = (pairing w).1
        then    let other := (pairing (Classical.choose hxf)).2
                if other ∈ hist then spam else other
        else    if hxs : ∃ w : win_sets, last = (pairing w).2
                then    let other := (pairing (Classical.choose hxs)).1
                        if other ∈ hist then spam else other
                else spam
    | [] => spam
```

If the adversary played one tile of the pair, this strategy plays the other. If not, playability of positional games ensures we may "spam" arbitrary (but unspecified) legal moves. Finally,

we state the following theorem, expressing that if a pairing exists, by playing according to the pairing strategy, the second player may win the game or lead it to a draw.

```
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
```

Tic-Tac-Toe

Finally, we discuss d-dimensional Tic-Tac-Toe on a cubical grid of side-length n, which in turn requires discussing combinatorial lines, since the winning sets will be the combinatorial lines of this cube.

▶ **Definition 8** (combinatorial lines). *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) (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) (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)
```

In order to prove the our main result on pairing strategies in Tic-Tac-Toe, we derive the following bounds, using Mathlib's API for double counting:

```
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. The latter of the above lemmata shows that the conditions to Hall's theorem, as it's stated in Mathlib, are satisfied. Piecing things together, we may conclude with:

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

4 Further work

The formalism we developed is in no way an accomplished one. One may, for example, experiment with restricting strategies to answer only to histories of moves in which the game hasn't ended. 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.

There are many more games to be modeled and reasoned about then those we presented, our favorite examples being Shannon's switching game [14] and Cops and robbers (on a graph, for instance) [2]. Violeta Hernández Palacios is currently leading work in the Mathlib community in making Conway's combinatorial games a smooth formal foundation for such concrete games.

References

1 Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

2 Anthony Bonato. *The game of cops and robbers on graphs*. American Mathematical Soc., 2011.

3 Charles L Bouton. Nim, a game with a complete mathematical theory. *Annals of mathematics*, 3(1/4):35–39, 1901.

4 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 73–78. Springer, 2009.

5 John Horton Conway. On numbers and games (1976). *Benutzte Ausgabe: Über Zahlen und Spiele. Übersetzt von Brigitte Kunisch. Braunschweig/Wiesbaden: Vieweg*, 1983.

6 Christoph Dittmann. Positional determinacy of parity games. *Available at www. isa-afp. org/browser_info/devel/AFP/Parity_Game/outline. pdf*, 2016.

7 David Gale. A curious nim-type game. *The American Mathematical Monthly*, 81(8):876–879, 1974.

8 David Gale. The game of hex and the brouwer fixed-point theorem. *The American mathematical monthly*, 86(10):818–827, 1979.

9 David Gale and Frank M Stewart. Infinite games with perfect information. *Contributions to the Theory of Games*, 2(245-266):2–16, 1953.

10 Alena Gusakov, Bhavik Mehta, and Kyle A Miller. Formalizing hall's marriage theorem in lean. *arXiv preprint arXiv:2101.00127*, 2021.

11 Dan Hefetz, Michael Krivelevich, Miloš Stojaković, and Tibor Szabó. *Positional games*, volume 44. Springer, 2014.

12 Sebastiaan JC Joosten. Gale-stewart games. 2024.

13 Deborah Kent and Matt De Vos. *Game theory: a playful introduction*. American Mathematical Society, 2017.

14 Alfred Lehman. A solution of the shannon switching game. *Journal of the Society for Industrial and Applied Mathematics*, 12(4):687–725, 1964.

15 Isabel Longbottom. Combinatorial game theory in lean. 2019.

16 Sven Manthe. A formalization of borel determinacy in lean, 2025. URL: `https://arxiv.org/abs/2502.03432`, `arXiv:2502.03432`.

17 Donald A Martin. A purely inductive proof of borel determinacy. *Recursion theory (Ithaca, NY, 1982)*, 42:303–308, 1985.

18 The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3372885.3373824`.

19 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021.

20 Ulrich Schwalbe and Paul Walker. Zermelo and the early history of game theory. *Games and economic behavior*, 34(1):123–137, 2001.

21 Ernst Zermelo. Über eine anwendung der mengenlehre auf die theorie des schachspiels. In *Proceedings of the fifth international congress of mathematicians*, volume 2, pages 501–504. Cambridge University Press Cambridge, 1913.