COMBINATORIAL GAME THEORY IN LEAN

ISABEL LONGBOTTOM

Abstract. This report outlines the results of and problems with a project to formalise the basic theory of combinatorial games in the functional programming language Lean. Combinatorial games were defined, along with addition and negation operations on games. An equivalence relation on games was also defined, under which combinatorial games form an abelian group. The required lemmas to prove that this is an abelian group were stated, although not all of the required proofs were completed. Major problems encountered included difficulties demonstrating that various relevant recursive functions were well-defined, and issues with formulating the definition of addition in a way which made the required proofs easy to construct.

1. Introduction

1.1. **A Background on Combinatorial Game Theory.** In the theory, a combinatorial game is a two player, deterministic game with complete information. A game must eventually end, draws are not possible, and the first player to be unable to make a legal move loses. We denote the two players by Left and Right. A game has some number of *left options*, comprising the allowable moves for Left, and some number of *right options*, the allowable moves for Right. Each of these options is itself a game, again described by the allowable moves for Left and Right. A game is uniquely determined by its left and right options; as such, a game can be defined mathematically as an ordered pair of (indexed) sets of games. That is, if L, R are sets of games, then $G := (L, R)$ is the game whose left options come from L and whose right options come from R. The requirement that all games must end is known as the *Descending Game Condition*. This condition states that there is no infinite sequence of games such that each term of the sequence is a left or right option of the previous term.
The notation

$$G = (\{G^{L_i} | i \in G_L\}, \{G^{R_j} | j \in G_R\})$$

can be used for a game G with left options indexed by G_L and right options indexed by G_R. Note that the left or right options of G may be empty.
From the above mathematical definition of a game, an induction principle on games can be obtained, called *Conway Induction*. This is stated as follows:

Theorem. *Suppose P is a property which a given game may or may not have. If a game G has property P whenever all left and right options of G have property P, then all games have property P.*

This can be proven directly from the definition of a game and the descending game condition.
Armed with this definition and the induction principle, an additive structure can be defined on combinatorial games such that they form an abelian group. To define the sum of

Date: May 2019.

two games A and B, imagine placing them side by side on a table. When it is Left's turn to move, she may choose to move either in game A or in game B, but not both. The sum of two games G and H can thus be defined algebraically as follows:

Definition. *If*

$$G = (\{G^{L_i} | i \in G_L\}, \{G^{R_j} | j \in G_R\})$$
$$H = (\{H^{L_i} | i \in H_L\}, \{H^{R_j} | j \in H_R\})$$

then the sum is given by

$$G+H := (\{G^{L_i}+H | i \in G_L\} \cup \{G+H^{L_i} | i \in H_L\}, \{G^{R_j}+H | j \in G_R\} \cup \{G+H^{R_j} | j \in H_R\}).$$

Note that the left options here are indexed by the union $G_L \cup H_L$ and similarly the right options are indexed by $G_R \cup H_R$.

A game can also be negated by switching its left and right options and negating all of its options. Algebraically, this is:

Definition. *If* $G = (\{G^{L_i} | i \in G_L\}, \{G^{R_j} | j \in G_R\})$*, the negative game is defined as*

$$-G := (\{-G^{R_j} | j \in G_R\}, \{-G^{L_i} | i \in G_L\}).$$

It can be shown using Conway Induction that this definition of addition is commutative and associative.

An order on games based on who has a winning strategy can also be defined. A player (Left or Right) *wins* a game if it is her opponent's turn and her opponent has no valid moves. That is, Left wins if it is Right's turn and the set of Right options is empty, and analogously for Right. Also, Left (resp. Right) has a *winning strategy* in some game G if there exists a sequence of valid moves for Left such that no matter which move Right selects each turn, Left will win. That is, there exists a left option G^L of G such that in every right option of G^L, Left can win if she goes first — so Left can win G if she goes first — and for every right option G^R of G, there exists some left option of G^R in which Left can win if she goes second — so Left can win G if she goes second.

Four possible outcome classes of a game G can now be defined. G *is a zero game* (written as $G = 0$) if there is a winning strategy for the second player no matter who starts; similarly, define G as being *fuzzy to zero* (or just *fuzzy*) if there is a winning strategy for the first player, no matter who starts. G is *positive* (written $G > 0$) if there is a winning strategy for Left (no matter who starts) and G is *negative* ($G < 0$) if there is a winning strategy for Right no matter who starts. It can be shown that for any two-person zero sum game these cases are exhaustive, and disjoint. Also, each of these properties can be defined recursively. For example, a game G is a zero game if each of its left options has a winning strategy for Right, and each of its right options has a winning strategy for Left. So a game is a zero game exactly when each of its left options is either fuzzy or negative, and each of its right options is either fuzzy or positive. (Note that once the game has moved to either a left or right option, the player whose turn it is has been fixed, since alternating turns are enforced. Hence in each left option it is sufficient to require that there be a winning strategy for Right if Right goes first, and analogously for the left options.)

Similar definitions can be made for negative, positive, and fuzzy games. It can be shown that these relations define an order on combinatorial games, but this was not a focus for this project. We were instead interested in defining the equivalence relation on games given by writing $G \cong H$ if $G - H$ is a zero game, using the definitions of addition and negation above. It can be shown that this is in fact an equivalence relation, that the

addition defined above respects the equivalence classes generated by this relation, and that modulo this equivalence relation combinatorial games form an abelian group under the inherited addition operation. Note that in the theory, commutativity and associativity of the operation are inherited from commutativity and associativity on combinatorial games without the relation. Introducing the relation makes $-G$ a true inverse for G. That is, this completes the group structure by providing inverses.
One final thing to note is that this is the notion of equivalence we want for games, since it interacts well with the four possible outcome classes (positive, fuzzy, negative, and zero). It can be shown, for example, that if you add a zero game to any other game then you preserve its outcome class and similarly if H is equivalent to K then the sums $G + H, G + K$ always have the same outcome.

1.2. **A Background on Dependent Type Theory and Lean.** What follows is a brief description of the functional programming language Lean in which this project was conducted, and its underpinning dependent type theory system.
Lean was developed as a tool for checking the correctness of program optimisation, but has since come into use as an interactive theorem proving language for mathematics. The power of the language comes from the fact that it can be used both to write proofs interactively, and to write automated tools (called *tactics*) which can then be used to assist proof-writing. The language is concerned both with generating proofs and with checking them for correctness. By keeping these two parts of its machinery distinct, and keeping the section used for checking the correctness of proofs minimal, a high level of confidence in its results can be ensured.
Where much of normal mathematics is based on the Zermelo-Fraenkel axioms of set theory, Lean is instead constructed on the alternate axiomatic basis of dependent type theory. Instead of having sets containing elements, we have *types* and *terms* of a type. We write `a : A` to mean `a` is a term of type `A`. Everything in Lean has exactly one type, including types themselves; this means, for example, that to use a natural number as a real number, we require a type coercion from one into the other (this corresponds to a set inclusion in set theory).
In order to formalise mathematical statements in Lean, the type Prop was introduced. Like any other type, this type can have terms. A term of type Prop (that is, a statement) is considered to be *true* if it is inhabited (if there exists a term of that type) and *false* if it is not inhabited. So a proof of a proposition is provided by constructing a term of the correct type. With this interpretation of elements of a type, a function from one proposition to another is simply a function which takes a term of one type (that is, a proof of one of the statements) and produces a term of the second type (that is, a proof of the other statement). This means that for `A B : Prop`, a function `f : A → B` is equivalent to a proof of the implication $A \implies B$.
We can similarly transfer many of the standard logical concepts into this context, such as $\vee, \wedge, \iff$ and so on. Note that unlike the Zermelo-Fraenkel system, dependent type theory is not built on first order logic as a metatheory. Instead, we model propositional logic internally, so that the above equivalence allows us to define logical implications in terms of types. In this context, checking a proof for correctness means checking that the proof term is of the required type. In general this process is substantially easier than constructing the proof term, and easy to do correctly.
We now consider some specific parts of Lean which were particularly relevant to this project.

When making an inductive definition in Lean, several auxiliary functions are automatically generated as part of the compilation of the definition. This usually includes, among other things, an induction principle. A simple example is as follows: we make the definition

```
inductive countable : Type
| base : countable
| next : countable → countable
```

(which of course is just the natural numbers renamed) and then using `#print prefix countable`, we can see the related functions that have been created. In the context of this project, we are interested in three of these functions:

```
countable.sizeof : countable → ℕ
countable.next.sizeof_spec : ∀ (a : countable),
        countable.sizeof (countable.next a) = 1 + sizeof a
countable.rec : Π {C : countable → Sort l},
  C countable.base → (Π (a : countable),
  C a → C (countable.next a)) → Π (n : countable), C n
```

Here `countable.rec` is the recursion principle for this inductive type, and `countable.sizeof` is an automatically generated size function which in some sense measures how far from the base case a specific term of this inductive type is.
The corresponding functions for combinatorial games played an important role in many of the proofs constructed as part of this project.
Finally, we discuss well-founded recursion in Lean. In order to construct a recursive definition (on an inductive type), a proof that the recursion is well-founded is required. This generally amounts to showing that the recursion is decreasing. That is, proving that the value of the recursive function on any specific (finite) input can be calculated by a terminating process (in a finite number of applications of the recursive function). Lean can often infer this proof on standard inductive types. The implementation of this can be explained in a simplified form as follows: if we have a recursive definition

```
def f : countable → countable
def g : countable → ℕ

def rec : countable → ℕ
| base := 0
| (next a) := g a + rec (f a)
```

then to show that this recursive definition is decreasing and well-founded, we are required to show that `f a` has a smaller size than `next a`. The means we have of comparing these sizes is to apply the automatically-generated `sizeof` function. Formally, we must prove that

```
countable.sizeof (f a) < countable.sizeof (next a)
```

for every `a : countable`. This inequality of course depends on the function `f`. For example, if we define

```
def f (a : countable) : countable := next a
```

then the inequality we want to prove becomes

```
countable.sizeof (next a) < countable.sizeof (next a)
```

which is clearly always false. This makes sense, as in the definition of `rec` we have defined

```
rec (next a) := g a + rec (next a)
```

which is clearly not a sensible (or valid) definition.

1.3. **Project Summary.** We attempted to formalise the definition of a combinatorial game in Lean. We defined combinatorial games, their addition and negation, and the four possible outcome classes (positive, negative, fuzzy, or zero). We were then able to construct the equivalence relation on games which is used to define the abelian group of games.
Apart from proving a couple of important lemmas, we were able to prove that the relation we defined was an equivalence realtaion, and were then able to prove that the quotient by this equivalence relation is an abelian group. We were also able to prove several useful lemmas regarding the outcome classes.
Proving that the quotient was an abelian group was substantially more difficult than anticipated. This was because our implementation of a game in Lean required that the left and right options be ordered in some deterministic way, and commuting two games in a sum or changing the order of addition reorders the left and right options in the resulting game. So in the context of our definition, addition of games was not commutative or associative until after applying the equivalence relation described above. This meant that many of the simple proofs used in the mathematical theory did not transfer to this context.

2. A First Attempt at Defining Combinatorial Games

2.1. **Definition of a Game.** Our first working attempt at defining a combinatorial game was to define a game inductively as two lists of games, representing the left and right options. We used:

```
inductive game₁ : Type
| intro : Π L R : list game₁, game₁

def game₁.L : game₁ → list game₁
| (game₁.intro L R) := L
def game₁.R : game₁ → list game₁
| (game₁.intro L R) := R
```

This definition had several problems:

- an induction principle on games (equivalent to Conway Induction[1]) was not generated automatically, so this was something we had to prove by hand
- reordering a list does not preserve equality, so the definition of addition of games here was not quite associative, since performing addition operations in a different order caused the left and right options of the result to be reordered within the list

[1]In fact, an induction principle was generated automatically. However, it was difficult to decipher, and did not seem to be equivalent to Conway Induction. As such, it was not useful. See section 2.3.

- in general, a proof that a given recursive definition was decreasing and well-founded could not be generated automatically, so had to be constructed manually for any recursive definition on games.

The final issue here was the most difficult to overcome, and was the eventual reason this definition was abandoned for a more generalised version. A discussion of this follows.

2.2. **Well-Founded Recursion.** With the above definition, Lean was unable to automatically generate a proof of well-founded recursion for any of the inductive definitions made on `game₁` (including addition, negation, Conway Induction, and the definition of a zero game). We use addition to exhibit the problem and the workaround we constructed. We would like to define addition as

```
def add : game₁ → game₁ → game₁
| G1 G2 := intro
(list.map (add G1) G2.L ++
                list.map (λ (g : game), add g G2) G1.L)
(list.map (add G1) G2.R ++
                list.map (λ (g : game), add g G2) G1.R)
```

However, when we make this definition, it fails to compile with the error message

```
failed to prove recursive application is decreasing, well
founded relation.
```

One solution to this problem is to append the block

```
using_well_founded
{dec_tac := my_dec_tac,
 rel_tac := λ _ _, `[exact ⟨_, measure_wf sizeof⟩]}
```

to this definition (and any other where well-founded recursion must be proved manually). This tells Lean to use the inferred `sizeof` instance for `game₁` for measuring what it means for the recursive application to be decreasing, and the tactic `my_dec_tac` to construct the proof term. We can then prove some appropriate lemmas and define `my_dec_tac` to apply them. By at first defining `my_dec_tac` to do nothing, we see that the proof term we require takes the form

```
⊢ has_well_founded.r ⟨G1, g⟩ ⟨G1, G2⟩
```

This is not a particularly enlightening goal, but using `dsimp` to expand several definitions related to well-founded recursion, we can eventually reduce the goal to something similar to

```
G1 G2 a : game₁,
⊢ 1 + game₁.sizeof G1 + game₁.sizeof a
                < 1 + game₁.sizeof G1 + game₁.sizeof G2
```

So, we must prove `game₁.sizeof a < game₁.sizeof G2`. This turns out to be impossible, since we have no hypothesis in the local context relating `a` to `G2`, and this inequality certainly doesn't hold for arbitrary games `a, G2`. Examining our definition of addition, we see that the variable `a` was introduced as a placeholder in one of the functions being mapped across a list, so represents an element of one of `G2.L, G1.L, G2.R, G1.R`. Specifically, an error message is being generated at the first recursive application of `add`

since this is the first place we must generate a proof of well-founded recursion, so in the goal above `a` represents an element of `G2.L`.
We now need two things to be able to prove well-foundedness:

- a proof that if `a ∈ G.L` then `game₁.sizeof a < game₁.sizeof G` and equivalently for the case where `a` is a right option of `G`; and
- some way to force the hypothesis `a ∈ G.L` into the local context so that we can apply our lemma.

The first of these is quite straightforward; we can prove

```
lemma sizeof_decr {g : game₁} {L : list game₁}
(h : g ∈ L) : game₁.sizeof g < sizeof L := sorry
```

by induction on the list `L`, and then this can be applied to both the left and right cases since `game₁.sizeof (intro L R) := 1 + sizeof L + sizeof R`.
The second is more difficult. The solution we found was to define a new function to use in place of `list.map` in the definition of addition. We defined

```
def map_mem {α β : Type*} :
Π (L : list α), (Π (x : α), (x ∈ L) → β) → list β
| [] f := []
| (y :: ys) f := (f y (list.mem_cons_self y ys)) ::
        map_mem ys (λ (x : α) (mem : x ∈ ys),
                        f x (list.mem_cons_of_mem y mem))
```

This function gives the same output as `list.map`, but its function argument has a different type. By making one of the arguments of the mapping function be a statement about membership in the list being mapped over, we force this hypothesis into the local context of any function definition in which `map_mem` is used. By redefining addition in terms of `map_mem`, we were thus able to modify the tactic state handed to `my_dec_tac` to include the appropriate hypothesis `a ∈ G1.L, a ∈ G2.L, a ∈ G1.R` or `a ∈ G2.R`. This allowed us to construct a working definition of addition.
However, this was by no means a complete solution to the problem of proving well-founded recursion. As we defined further recursive functions on `game₁`, similar but slightly different tactics were required to demonstrate well-founded recursion, with several addition lemmas about `game₁.sizeof, list.sizeof` also needed. This made many of the definitions slightly harder to work with, since instead of using functions from mathlib such as `list.map`, about which some things have already been proven, we were required to define new functions like `map_mem`.

2.3. **The Inferred Induction Principle Versus Conway Induction.** It is worth embarking on a brief detour at this point to consider our implementation of Conway Induction for this definition of a combinatorial game. Looking at the automatically generated induction principle for `game₁`, we have

```
game₁.rec : Π (C : game₁ → Sort l),
        (Π (L R : list game₁), C (game₁.intro L R))
                → Π (x : game₁), C x
```

This effectively says that if we have any proposition about `game₁` (or function with domain `game₁`) and we can prove that it is true (or well-defined, in the function case)

of the game formed from any two lists of games, then it is true of all games. This is true, but it is not a particularly useful induction principle; we could construct a short proof ourselves as follows:

```
lemma game₁.rec : Π (C : game₁ → Sort l),
        (Π (L R : list game₁), C (game₁.intro L R))
                → Π (x : game₁), C x :=
begin
  intros C h x,
  induction x with L R,
  exact h L R,
end
```

We note that the induction principle generated here is not just different to Conway Induction, it is a weaker statement and will be substantially less useful in later proofs. So, we must define and prove Conway Induction by hand.
We use the following definition:

```
def Conway_ind (P : game₁ → Prop)
(w : Π (g : game₁), ((Π l : game₁, (l ∈ game₁.L g → P l))
        ∧ (Π r : game₁, (r ∈ game₁.R g → P r))) → P g)
: Π G, P G
| (intro L R) :=
begin
  apply w,
  split,
  { intros l H,
    exact Conway_ind l, },
  { intros r H,
    exact Conway_ind r, }
end
```

and we note that this must be a `def` not a `lemma` or `theorem` as this allows us to construct the proof inductively. As with our definition of addition, Lean requires us to demonstrate that the recursion in this definition is well-founded; the proof is similar to that of addition, and the details are uninteresting. The significance of this example stems from the fact that this proof clearly depends substantially on the fact that the recursion is well-founded. For example we could attempt to construct the proof as

```
def Conway_ind' (P : game₁ → Prop)
(w : Π (g : game₁), ((Π l : game₁, (l ∈ game₁.L g → P l))
        ∧ (Π r : game₁, (r ∈ game₁.R g → P r))) → P g)
: Π G, P G
| (intro L R) :=
begin
  exact Conway_ind' (intro L R),
end
```

and this is clearly not a valid proof. This generates a similar error message about a failure to show decreasing well-founded recursion, but with the goal as `⊢ 1 < 1` instead of

something similar to what was generated when we were constructing the proof for addition. This is reassuring in that the goal here is false, so this incorrect proof would likely not be able to be made acceptable, and also emphasises the importance of demonstrating that any use of recursion is well-founded and decreasing.

3. A Better Attempt

3.1. **Arbitrary Indexing.** Many of the problems we encountered in our first attempt at defining combinatorial games were caused by the fact that most of the things we needed, such as an induction principle, could not be generated automatically. This is somewhat unusual; in general, we should expect that in the process of compiling the definition of an inductive type, a (useful) induction principle for that type should be generated. In the hope of being able to automatically generate a version of Conway Induction, we made a more general definition of a game

```
inductive game : Type (u+1)
| intro : Π (l : Type u) (r : Type u)
                (L : l → game) (R : r → game), game
```

In this definition, the left options of a game `intro l r L R` are indexed over the type `l`, with the function `L` mapping from the indexing set to the games which are the left options, and similarly the right options are indexed over the type `r`. Requiring `l, r` to be subsets of the natural numbers recovers something equivalent to our previous definition, where the left and right options were elements of a list (with the empty list corresponding to the empty set, and finite length lists corresponding to finite subsets of $\mathbb{N}$). However, in this new definition the left options could be indexed by, say, $\mathbb{R}$, which does not correspond to a (countable) list. Hence this is a true generalisation of our previous definition.
With this definition, Lean was able to infer an induction principle, of the form

```
game.rec : Π {C : game → Sort l},
  (Π (l r : Type u) (L : l → game) (R : r → game),
     (Π (a : l), C (L a)) → (Π (a : r), C (R a))
     → C (game.intro l r L R)) → Π (n : game), C n
```

which is exactly the form Conway Induction takes in this arbitrary indexing context. This makes the more general definition seem more promising than our first attempt.

3.2. **Positive, Negative, Fuzzy, and Zero Games.** We wish to define each of the four outcome classes recursively. We note that in such a definition, each of these outcome classes would depend on at least two of the others, so we would have to define these four properties in a mutually recursive way. Lean has the syntax `mutual def` for this purpose. However, if we use this method then the resulting function we generate will likely not be particularly nice, and the Lean compiler may not be able to generate a proof that the recursion is well-founded. We can instead define the statements `positive G ∨ fuzzy G` (that is, Left has a winning strategy if she goes first) and `negative G ∨ fuzzy G` (that is, Right has a winning strategy if he goes first) in a mutually recursive manner, and then define the four outcome classes we want in terms of these two things. We make the definition

```
def is_pos_fuzz_is_neg_fuzz (x : game) : Prop × Prop :=
begin
  induction x with xl xr xL xR IHxl IHxr,
  dsimp at *,
  exact (∃ i : xl, ¬(IHxl i).2, ∃ i : xr, ¬(IHxr i).1)
end

def is_pos_fuzz (x : game)
:= (is_pos_fuzz_is_neg_fuzz x).1
def is_neg_fuzz (x : game)
:= (is_pos_fuzz_is_neg_fuzz x).2
```

This definition can be interpreted as saying that the game `G` is positive or fuzzy if there exists some left option where Left has a winning strategy if Right goes first, and negative or fuzzy if there exists some right option where Right has a winning strategy if Left goes first, which is the correct definition. As with the definition of addition (discussed in section 3.3), this definition is not actually recursive except in that it uses the induction principle for games to generate `IHxl, IHxr`, so we have no issues with well-founded recursion. We then define the compound statements:

```
def is_zero : game → Prop
| G := (¬ is_pos_fuzz G) ∧ (¬ is_neg_fuzz G)
def is_fuzz : game → Prop
| G := is_pos_fuzz G ∧ is_neg_fuzz G
def is_pos : game → Prop
| G := (is_pos_fuzz G) ∧ ¬ (is_fuzz G)
def is_neg : game → Prop
| G := (is_neg_fuzz G) ∧ ¬ (is_fuzz G)
```

to get the four outcome classes we want. Using these definitions, and proving several intermediate lemmas, we can formulate each of these statements in terms of the other three, so that when working with the four outcome classes we do not require recourse to `is_pos_fuzz_is_neg_fuzz`.

This allows us to effectively define each outcome class in terms of universal and existential statements about the outcome classes of the left and right options — that is, in the form we wish to use them — by proving `@[simp]` lemmas for each outcome class, but without defining them in the first place as a complicated `mutual def` for which we must prove well-founded recursion by hand. This also produces much simpler proofs that the four outcome classes are exhaustive and disjoint than we had previously with the definitions for `game₁`.

3.3. **Addition and Negation in this Context.** With this definition of a combinatorial game, we were able to define negation without providing an explicit proof of well-founded recursion. We defined

```
def neg : game → game
| ⟨l, r, L, R⟩ := ⟨r, l, λ i, neg (R i), λ i, neg (L i)⟩
```

and the Lean compiler was able to automatically construct a proof of well-founded recursion. We were also able to prove that negating a game twice returns the original game directly from the definition and using Conway induction.

Defining addition was more difficult. We initially constructed the following definition using the induction principle on `game`:

```
def add (x y : game) : game :=
begin
  induction x with xl xr xL xR IHxl IHxr,
  induction y with yl yr yL yR IHyl IHyr,
  dsimp at *,
  refine ⟨xl ⊕ yl, xr ⊕ yr, sum.rec _ _, sum.rec _ _⟩,
  { exact IHxl },
  { exact IHyl },
  { exact IHxr },
  { exact IHyr }
end
```

This definition is only recursive in that it uses the recursor on `game` to generate the functions IHxl, IHxr, IHyl, and IHyr. It doesn't actually call itself in its definition. This means that no proof of well-founded recursion is required. However, this definition of addition turns out to not be very useful. Consider the following lemma:

```
lemma is_zero_sub {G : game} : is_zero (G - G) := sorry
```

The standard mathematical proof of this fact proceeds as follows:

Proof. We show that the second player has a winning strategy. Observe first that the valid moves for Left in $-G$ match the valid moves for Right in G, and vice versa. Without loss of generality, suppose Right goes first. If Right has a valid move in G, say G^R, then $-G^R$ is a left option in $-G$. Left can thus choose this move, and the players reach the game $G^R - G^R$, with it being Right's turn. By (Conway) induction, this is a zero game, so Left — being the second player — has a winning strategy. This is true no matter which right option of G Right chooses as his move, so Left can win no matter which right option of G Right picks. Similarly, if Right instead chooses to move in $-G$, taking the option $-G^L$, then G^L is a left option of G, so Left has a valid move. This puts the players in the game $G^L - G^L$, which again has a winning strategy for the second player, Left. So in either case, Left has a winning strategy. The proof is similar if Left goes first.
Intuitively, the second player may win by always making the same move as the first player just made, in the other game. Then the two boards will always match, so the second player can continue in this manner, and will never be lost for a move so cannot lose. □

It should be possible to prove this lemma using our definitions in a similar manner, using the idea that the second player always copies the first player's move, and considering 4 analogous cases (corresponding to Left playing first in G, Left playing first in $-G$, Right playing first in G, and Right playing first in $-G$), applying Conway Induction in each. However, this approach encounters a substantial roadblock when we try to unfold the definition of addition. We would like to be able to rewrite the addition in some way which makes it clear that the left options of $G - G$ take the form $G - G^L, G^L - G$ and similarly for the right options. To do this, we define a lemma:

```
@[simp] lemma add_def {xl xr xL xR yl yr yL yR} :
  add (intro xl xr xL xR) (intro yl yr yL yR) =
    intro (xl ⊕ yl) (xr ⊕ yr)
    (λ xy, sum.cases_on xy
     (λ xy : xl, add (xL xy) (intro yl yr yL yR))
     (λ xy : yl, add (intro xl xr xL xR) (yL xy)))
    (λ xy, sum.cases_on xy
     (λ xy : xr, add (xR xy) (intro yl yr yL yR))
     (λ xy : yr, add (intro xl xr xL xR) (yR xy)))
:= sorry
```

and then instead of using `dsimp[add]` to unfold the definition of addition, we can use `rw add_def` to get the result in a more useful form. However, we were not able to complete the proof of this lemma. This is because it is not actually true, and suggests that this definition of addition is not constructing the left and right options in the way we want. So we require a different definition of addition.
If we try to define addition directly in the form we want, as something like:

```
def add' : game → game → game
| (intro xl xr xL xR) (intro yl yr yL yR) :=
⟨xl ⊕ yl, xr ⊕ yr,
    (λ xy, sum.cases_on xy
     (λ xy : xl, add' (xL xy) (intro yl yr yL yR))
     (λ xy : yl, add' (intro xl xr xL xR) (yL xy)))
    (λ xy, sum.cases_on xy
     (λ xy : xr, add' (xR xy) (intro yl yr yL yR))
     (λ xy : yr, add' (intro xl xr xL xR) (yR xy)))⟩
```

then we encounter the same well-foundedness issues as in our definition `game₁`. We could potentially resolve these issues in a similar way as discussed above, however in this context the automatically generated `game.sizeof` function is not as useful, since it is defined as

```
game.intro.sizeof_spec : ∀ (l r : Type u)
        (L : l → game) (R : r → game),
                game.sizeof (intro l r L R) = 1
```

and so if we try to use this concept of size to demonstrate well-foundedness, we get the goal

```
⊢ game.sizeof (xL xy) < game.sizeof (intro xl xr xL xR)
```

or equivalently `⊢ 1 < 1`, which we cannot prove. Instead, we modify our original attempt at defining addition to be:

```
def add (x y : game) : game :=
begin
  induction x with xl xr xL xR IHxl IHxr generalizing y,
  induction y with yl yr yL yR IHyl IHyr,
  have y := intro yl yr yL yR,
  refine ⟨xl ⊕ yl, xr ⊕ yr, sum.rec _ _, sum.rec _ _⟩,
  { exact λ i, IHxl i y },
  { exact λ i, IHyl i },
  { exact λ i, IHxr i y },
  { exact λ i, IHyr i }
end
```

which compiles correctly. This definition is also mathematically correct, since we note that the lemma `add_def` is reflexively true under this definition of addition (that is, it can be proved by `refl`).

3.4. **The Equivalence Relation - Required Proofs.** We will now discuss the equivalence relation we wish to define on games so that they form an abelian group. We define

```
def equiv (G H : game) : Prop := is_zero (add G (neg H))
```

and we wish to prove several things about this definition:

(1) that it is an equivalence relation (that is, we must prove reflexivity, symmetry, and transitivity);
(2) that the addition operation we defined on games respects the equivalence classes under this relation; and
(3) that games form an abelian group under this relation.

We progressed to the point of stating and proving these facts, assuming one lemma which we were unable to prove. Reflexivity, symmetry and transitivity can be stated as follows:

```
theorem equiv_refl {G : game} : equiv G G := sorry

theorem equiv_symm {G H : game}
(h : equiv G H) : equiv H G := sorry

theorem equiv_trans {G H K : game}
(h₁ : equiv G H) (h₂ : equiv H K)
: equiv G K := sorry
```

Similarly, we can state the theorem we need to show that the definition of a addition on games extends to one on equivalence classes of games as follows:

```
theorem add_resp_equiv (G J H K : game) (h : equiv G H)
(k : equiv J K) : equiv (G + J) (H + K) := sorry
```

since together these two theorems prove that addition of equivalence classes (defined by choosing an element from each equivalence class, adding them, then taking the equivalence class of the result) is independent of the choice of representative element. We need to show a similar fact for negation, that is,

```
theorem neg_resp_equiv {G H : game}
(h : equiv G H)
:equiv (neg G) (neg H) := sorry
```

noting that this theorem can be made to follow from `neg (G + H) = neg G + neg H` and `neg (neg H) = H`, given a lemma of the form

```
theorem neg_zero_iff {G : game}
: is_zero G ↔ is_zero (neg G) := sorry
```

Finally, we must show that the equivalence classes form an abelian group under the inherited addition operation. This means proving commutativity, associativity, and the inverse property. However, we note that the inverse property is (almost) equivalent to reflexivity of the equivalence relation, so we need only demonstrate the other properties:

```
theorem add_zero {G : game} : equiv (G + 0) G
:= sorry

theorem zero_add {G : game} : equiv (0 + G) G
:= sorry

theorem add_assoc {G H K : game}
: equiv ((H + G) + K) (H + (G + K)) := sorry

theorem add_comm {G H : game}
: equiv (G + H) (H + G) := sorry
```

Looking at the above lemmas, we note that to prove most of them, we wish to be able to commute and reassociate addition inside `is_zero()`. For example, assuming reflexivity, we want to be able to rewrite the statement of commutativity as follows

```
is_zero((G + H) - (H + G)) ↔ is_zero((G - G) + (H - H))
```

and then use the fact that `G - G` and `H - H` are both zero games to prove the rearranged version on the right. This kind of rearrangement can then be used to prove many of the above lemmas.
Our general strategy was thus as follows:

- prove reflexivity, that is, that `G - G` is a zero game for any game `G`;
- prove some rearrangement lemmas, likely including one of the form

  `is_zero(G + H) ↔ is_zero(H + G)`

  and a corresponding lemma for associativity;
- prove that the sum of two zero games is a zero game.

We hoped to then apply these lemmas to all, or most, of the above. Proceeding in this way, we were able to prove reflexivity[2] by induction.

[2]The general strategy was the same as in the mathematical proof outlined in section 3.3, and as we expected, we were required to provide almost the same proof four times, once for each of the four cases.

3.5. **Rearrangement - Some Problems and Solutions.** We next proved some rearrangement lemmas. Our first approach was to prove the following by induction:

```
lemma is_zero_comm {G H : game} :
is_zero (G + H) ↔ is_zero (H + G) := sorry

lemma is_zero_assoc {G H K : game} :
is_zero (G + (H + K)) ↔ is_zero ((G + H) + K) := sorry
```

However this was difficult to do directly. To illustrate the problem, we note that in the forwards direction for commutativity, the induction hypothesis produced for the left options of `G` was of the form

```
∀ (a : G_l), is_zero (G_L a + H) ↔ is_zero (H + G_L a)
```

This is vacuously true, but not particularly useful, since neither side of this statement is true[3]. A more useful induction hypothesis involves a similar statement about negative or fuzzy games, perhaps something of the form

```
∀ (a : G_l), is_neg_fuzz (G_L a + H)
                ↔ is_neg_fuzz (H + G_L a)
```

In light of this, it should in theory be possible to prove the following lemma by induction, using each part of the induction hypothesis in the proof of at least one other part of the statement:

```
lemma outcomes_commute {G H : game} :
(is_zero (G + H) ↔ is_zero (H + G)) ∧
(is_fuzz (G + H) ↔ is_fuzz (H + G)) ∧
(is_neg (G + H) ↔ is_neg (H + G)) ∧
(is_pos (G + H) ↔ is_pos (H + G)) := sorry
```

However, this would not be a particularly nice approach, since we would need to produce proofs for approximately 16 different cases. We note that we originally defined each of these four outcome classes in terms of just two recursively defined properties, `is_neg_fuzz` and `is_pos_fuzz`, in order to avoid a similar kind of mutual recursion. By reframing this lemma in terms of only two predicates instead of four, we can cut down the number of cases by at least a factor of two. We state our reformulated rearrangement lemma as:

```
lemma neg_fuzz_pos_fuzz_comm {G H : game} :
(is_neg_fuzz (G + H) ↔ is_neg_fuzz (H + G)) ∧
(is_pos_fuzz (G + H) ↔ is_pos_fuzz (H + G)) := sorry
```

We were able to prove this lemma via a straightforward induction. We were similarly able to prove a form of associativity using this method. The proofs of the statements we wanted about `is_zero` were then direct applications of these lemmas, since we defined `is_zero` as a simple compound logical statement in terms of `is_pos_fuzz`, `is_neg_fuzz`.

[3]Since `G + H` is a zero game, each of its left options is fuzzy or negative. Hence `G_L a + H` is fuzzy or negative, so cannot be a zero game.

Leaving aside for now the proof that the sum of two zero games is a zero game, we wish to apply our rearrangement lemmas to, say, the proof that addition commutes on the quotient. As mentioned above, we want to prove that

```
is_zero((G - G) + (H - H)) → is_zero((G + H) - (H + G))
```

since the right hand side here is the definition of `equiv (G + H) (H + H)`. However, only using the two rearrangement lemmas above, this is impossible. The problem is that when using commutativity and associativity under ordinary circumstances, we implicitly assume that the notion of equality we are working with is an equivalence relation. This means that having proven, say, $A + B = B + A$, we automatically know that $(A + B) + C = (B + A) + C$. However, in this context, we wish to use these rearrangement lemmas to show that `is_zero` defines an equivalence relation. So, for example

```
is_zero((G + H) + X) ↔ is_zero((H + G) + X)
```

does not follow directly from the rearrangement lemmas we already have. Moreover, using the two rearrangement lemmas we proved, it is not possible to achieve all possible permutations of an arbitrary string of addition operations. The solution we discovered to this problem was to state and prove two additional rearrangement lemmas, namely

```
lemma neg_fuzz_pos_fuzz_zoom_comm {G H X : game} :
(is_neg_fuzz ((G + H) + X) ↔ is_neg_fuzz ((H + G) + X)) ∧
(is_pos_fuzz ((G + H) + X) ↔ is_pos_fuzz ((H + G) + X))
:= sorry

lemma neg_fuzz_pos_fuzz_zoom_assoc {G H K X : game} :
(is_neg_fuzz ((G + (H + K)) + X)
        ↔ is_neg_fuzz (((G + H) + K) + X)) ∧
(is_pos_fuzz ((G + (H + K)) + X)
        ↔ is_pos_fuzz (((G + H) + K) + X)) := sorry
```

and the corresponding statements for `is_zero`. Using all four of these rearrangement lemmas, and the statement

```
lemma add_zero_still_zero {G H : game} (hG : is_zero G) :
is_zero H ↔ is_zero (G + H) := sorry
```

we were able to prove all of the lemmas stated in section 3.4. This allowed us to construct the abelian group of games, and prove that it is an abelian group under the inherited addition and negation operations.

4. Persistent Problems

4.1. **Ordered Indexing and its Consequences.** In all of the valid definitions of a combinatorial game we were able to construct, we indexed the left and right options in some ordered manner, rather than representing the left (resp. right) options as a set with no further structure. This had some unfortunate arithmetic consequences. One of these was that addition as defined on games without the equivalence relation was not associative or commutative. To highlight this problem, we exhibit the proofs of associativity and commutativity in the unordered theoretical context, and demonstrate why they fail in the ordered context we implemented.

In the theoretical context, where the left and right options are each defined as sets, we have

$$\begin{aligned} G+H &= (\{G^{L_i}+H | i \in G_L\} \cup \{G + H^{L_i} | i \in H_L\}, \{G^{R_j}+H | j \in G_R\} \cup \{G + H^{R_j} | j \in H_R\}) \\ &= (\{G + H^{L_i} | i \in H_L\} \cup \{G^{L_i}+H | i \in G_L\}, \{G + H^{R_j} | j \in H_R\} \cup \{G^{R_j}+H | j \in G_R\}) \\ &= (\{H^{L_i}+G | i \in H_L\} \cup \{H + G^{L_i} | i \in G_L\}, \{H^{R_j}+G | j \in H_R\} \cup \{H + G^{R_j} | j \in G_R\}) \\ &= H+G \end{aligned}$$

with the reordering in the second line allowed because the union operator on sets is commutative, and the third equality by Conway Induction. The problem we encounter in our implementation of games comes in line 2. We can apply Conway Induction to commute the sums within individual left and right options, so it is possible to prove that $G+H$ and $H+G$ have the same left options as each other and the same right options as each other, but these options do not appear in the same order. The corresponding operation in our definitions to the set union here is either list concatenation (in `game₁`) or the direct sum of two types (in the arbitrary indexing context), neither of which are commutative.

One possible solution to this problem would be to apply an equivalence relation which made two games equal if the left options of one were some permutation of the left options of the other, and similarly for the right options. This would allow us to prove commutativity (and associativity, since the proof is similar and breaks in the same way) however it is not a particularly desirable solution. This equivalence relation would not give us an abelian group, since negation as defined previously would not work. Also, the equivalence relation defined in section 3.4 would allowed us to eventually prove commutativity and associativity anyway.

This leaves us in a somewhat frustrating position: the equivalence classes which form an abelian group in the theory still form an abelian group in this implementation, but a greater portion of the group structure depends on the equivalence relation, making several proofs more complicated.

4.2. **The Sum of Two Zero Games.** In section 3.5, we stated the following lemma about zero games:

```
lemma add_zero_still_zero {G H : game} (hG : is_zero G) :
is_zero H ↔ is_zero (G + H) := sorry
```

This is the final part of this project to be done, and the only part of constructing the abelian group of games which we were unable to complete. In the theory of combinatorial games, the proof follows from the following theorem:

Theorem. *The sum of a zero game G and any other game H has the same outcome class as H.*

Proof. Whoever has a winning strategy in H may win $G+H$ by always playing in the following way. When it is her turn to move in $G+H$, the player should make a move in H according to her winning strategy in H, except when her opponent moves in G, in which case the player should reply in G according to the winning stategy for the second player in G. We note that since the player moves in G only in reply to her opponent's moves, she is always the second player in G, and so has a winning strategy there. Also, since the players alternate moves in G, they must also alternate moves in H, so in the sum play proceeds in H exactly as it does when we consider just the game H. This means

that the player will never be lost for a move in either game while her opponent still has a move in that game, so will win the sum. □

The fact that the statement of our lemma is an if and only if then proceeds from the fact that the four possible outcome classes are disjoint. We were unable to prove the lemma we want directly. We then tried unsuccessfully to prove each of the following formulations of similar lemmas:

```
lemma neg_fuzz_pos_fuzz_add_zero_iff
{G H : game} (h : is_zero G) :
(is_neg_fuzz H ↔ is_neg_fuzz (H + G)) ∧
(is_pos_fuzz H ↔ is_pos_fuzz (H + G)) := sorry

lemma neg_fuzz_pos_fuzz_add_zero
{G H : game} (h : is_zero G) :
(is_neg_fuzz H → is_neg_fuzz (H + G)) ∧
(is_pos_fuzz H → is_pos_fuzz (H + G)) := sorry
```

Our problems in each case can be summarised as follows:

- to prove the forwards direction of the implication, we needed the reverse direction of the implication to appear in the inductive hypothesis;
- the reverse direction of the implication was difficult to prove by induction, since the the proof in the theory doesn't construct a winning strategy in H given one in $G + H$, it instead uses the fact that the four possible outcome classes are disjoint to derive a contradiction if H has a different outcome from $G + H$.

This meant that in order to prove the forwards direction, we seemed to be required to prove both directions simultaneously by mutual induction, but to prove the reverse direction we needed to first prove the forwards direction separately. We were unable to resolve these difficulties.

5. Summary of Results

We were able to define combinatorial games in Lean, define the four possible outcome classes for games (positive, negative, fuzzy, and zero) and prove a comprehensive set of lemmas about these outcome classes. We were also able to define addition and negation operations on games.

We were able to define the equivalence relation needed to construct the abelian group of combinatorial games, and state the required lemmas for proving that this does, in fact, construct an abelian group under the inherited addition and negation operations and that all the required operations were well-defined. Using one lemma which we were unable to prove in Lean, we were able to prove all of these and thus to construct the abelian group of games.

6. References

[1] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem proving in Lean*. Microsoft Research, https://leanprover. github. io/tutorial/tutorial. pdf, 2015.

[2] Mario Carneiro. *Surreal numbers*. https://github.com/leanprover-community/mathlib/blob/master/src/set_theory/surreal.lean, 2019.

[3] John H Conway. *On numbers and games*. AK Peters/CRC Press, 1976.

[4] Dierk Schleicher and Michael Stoll. An introduction to conway's games and numbers. *Moscow Mathematical Journal*, 6(2):359–388, 2006.