# Combinatorial Game Theory in Lean

Isabel Longbottom

Australian National University

June 2019

# Mathematical Background

- $G = (\{G^{L_i} | i \in G_L\}, \{G^{R_j} | j \in G_R\})$
- A player loses if there are no valid moves (i.e. corresponding set is empty)
- Conway induction: 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 $P$.
- Addition:
  $G+H := (\{G^{L_i}+H, \ldots, G+H^{L_i}, \ldots\}, \{G^{R_j}+H, \ldots, G+H^{R_j}, \ldots\})$
- Negation: $-G := (\{-G^{R_j} | j \in G_R\}, \{-G^{L_i} | i \in G_L\})$
- Zero games: $G$ is a zero game if the second player has a winning strategy
- Equivalence relation: $G \cong H$ if $G - H$ is a zero game.

# Project Goals

- Define combinatorial games, addition, negation, etc.
- Prove that the relation defined on the previous slide is an equivalence relation
- Construct the quotient by this equivalence relation
- Show that addition and negation are well-defined on the quotient
- Show that the quotient is an abelian group under the inherited addition and negation

# Defining a Game

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

- General definition
- Allows arbitrarily many left and right options
- Useful induction principle (Conway induction) is automatically generated

```
protected eliminator 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 (intro l r L R)) → Π (n : game), C n
```

# Negation and addition

- Negation:

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

- Addition:

```
def add (x y : game) : game :=
begin
  induction x generalizing y,
  induction y,
  have y := intro y_l y_r y_L y_R,
  refine ⟨x_l ⊕ y_l, x_r ⊕ y_r,
             sum.rec _ _, sum.rec _ _⟩,
  { exact λ i, x_ih_L i y },
  { exact λ i, y_ih_L i },
  { exact λ i, x_ih_R i y },
  { exact λ i, y_ih_R i }
end
```

# Defining the Outcome Classes

- Each of the four outcome classes can be defined recursively, in a way that depends on some of the others
- E.g. $G$ is a zero game if each left option $G^{L_i}$ is negative or fuzzy, and each right option $G^{R_j}$ is positive or fuzzy
- To avoid mutual recursion, define two compound outcome classes

```
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
```

# Defining the Outcome Classes

```
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)
```

# The Equivalence Relation

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

- We want to prove the following:

```
lemma equiv.refl {G : game} :
is_zero (G - G) := sorry

lemma equiv.symm {G H : game}
(h : is_zero (G - H)) :
is_zero (H - G) := sorry

lemma equiv.trans {G H K : game}
(h₁ : is_zero (G - H))
(h₂ : is_zero (H - K)) :
is_zero (G - K) := sorry
```

# A Note on Indexed Sets

- The theory differs somewhat from our approach
- In theory, addition is commutative and associative before quotienting by the equivalence relation:

$$\begin{aligned} G + H &= (\{G^{L_i} + H, \ldots, G + H^{L_i}, \ldots\}, \{G^{R_j} + H, \ldots, G + H^{R_j}, \ldots\}) \\ &= (\{G + H^{L_i}, \ldots, G^{L_i} + H, \ldots\}, \{G + H^{R_j}, \ldots, G^{R_j} + H, \ldots\}) \\ &= (\{H^{L_i} + G, \ldots, H + G^{L_i}, \ldots\}, \{H^{R_j} + G, \ldots, H + G^{R_j}, \ldots\}) \\ &= H + G \end{aligned}$$

- Second line follows since these are sets, and doesn't follow in our construction using indexing over a type
- This proof was not possible in Lean, because this statement is not true when the options are indexed

# Rearrangement Lemmas

```
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

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

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
```

# Adding a Zero Game Preserves Outcome Class

This is the final lemma required:

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

- We should be able to prove this by showing the forwards direction of the implication for each of the four possible outcome classes
- Or (more briefly) by showing

```
is_zero G ∧ is_neg_fuzz H → is_neg_fuzz (G + H)
```

```
is_zero G ∧ is_pos_fuzz H → is_pos_fuzz (G + H)
```

# Example Proof - Addition Respects Equivalence Classes

```
lemma add_resp_equiv (G J H K : game)
(h : equiv G H) (k : equiv J K) :
equiv (G + J) (H + K) :=
begin
  dsimp [equiv] at *,
  rw [neg_distrib, is_zero_assoc,
      ← is_zero_zoom_assoc, is_zero_zoom_comm,
      ← is_zero_zoom_assoc, is_zero_zoom_comm,
      ← is_zero_assoc, is_zero_zoom_comm],
  exact (add_zero_still_zero h).1 k,
end
```

- After all the rewrites, the goal is:

```
⊢ is_zero (G+(-H)+(J+(-K)))
```