Research Practice I

Formalization of a Variant of the Kirby–Paris Hydra Game in Coq

John González-González[†] and Andrés Sicard-Ramírez[†]
jagonzale4@eafit.edu.co, asr@eafit.edu.co

[†]School of Applied Sciences and Engineering, Universidad EAFIT

May 28, 2025

Abstract

The Kirby–Paris Hydra Game presents a fascinating paradox: despite appearing to expand exponentially during play, the game invariably terminates after a finite number of steps. This property relates to fundamental limits in formal systems, particularly Peano arithmetic. In this paper, we present a simplified formalization of the Hydra Game using the Coq proof assistant. We represent hydras as rose trees and develop a lexicographic termination proof based on counting nodes at each level. This representation allows us to establish that each game step produces a lexicographically smaller list, guaranteeing termination. We present the key theorems and lemmas of our formalization, explaining their intuitive meaning and formal significance within the proof structure. Our work demonstrates the application of interactive theorem proving to verify properties of combinatorial games with deep connections to mathematical logic, and provides an accessible case study of formal verification using the Coq proof assistant.
Keywords: Interactive theorem proving, Kirby–Paris Hydra Game, termination proofs, lexicographic ordering, Coq proof assistant, formal verification, rose trees

1 Introduction

The Kirby–Paris Hydra Game, introduced by Laurie Kirby and Jeff Paris in 1982 **?**, presents a fascinating paradox in combinatorial game theory. Despite appearing to grow exponentially during play, the game invariably terminates after a finite number of steps. This property relates to fundamental limits in formal systems, particularly Peano arithmetic, similar to Goodstein's theorem **?**.

In the Hydra Game, a player repeatedly "chops off" heads (leaf nodes) from a tree-structured hydra. After each chop, new heads grow according to specific rules, seemingly causing the hydra to expand out of control. Counterintuitively, any strategy—even deliberately choosing moves that maximize growth—inevitably leads to victory (complete removal of the hydra) in a finite number of steps.

The termination of this game cannot be proven within Peano arithmetic, making it a canonical example of a true mathematical statement whose proof requires stronger formal systems. This connects the Hydra Game to fundamental results in mathematical logic and the foundations of mathematics.

In this paper, we present a simplified formalization of the Hydra Game using the Coq proof assistant **?**. We represent hydras as rose trees and develop a lexicographic termination proof based on counting nodes at each level. This representation allows us to establish that each game step produces a lexicographically smaller list, guaranteeing termination.

The contributions of this paper include:

- A simplified representation of the Hydra Game suitable for formal verification
- A complete formalization and proof of termination in the Coq proof assistant
- A detailed explanation of the lexicographic ordering approach to proving termination
- Visual animations of the game dynamics to provide intuition for the formal model

Our work demonstrates the application of interactive theorem proving to verify properties of combinatorial games with deep connections to mathematical logic and provides an accessible case study of formal verification using the Coq proof assistant.

2 Background

2.1 The Kirby–Paris Hydra Game

The original Hydra Game, introduced by Kirby and Paris **?**, is played on a rooted tree where:

- A player repeatedly chooses a leaf node (a "head") to remove
- After each removal, depending on the structure of the tree around the removed leaf, new branches grow according to specific rules
- The player wins if they eventually remove the entire tree

The game appears to create an ever-expanding tree, yet Kirby and Paris proved that it always terminates. Moreover, they demonstrated that the proof of termination cannot be formalized within Peano arithmetic, connecting this combinatorial game to Gödel's incompleteness theorems.

In our simplified variant, we focus on a version where:

1. The player always repeatedly chooses a leaf node (a "head") to remove
2. After removing a leaf connected to node N, if N has a grandparent G, then n new leaves are added as children of G (where n is a fixed parameter), each identical to the removed leaf.
3. If the removed leaf was directly connected to the root, the game advances without adding new nodes

This deterministic variant preserves the essential properties of the original game while being more amenable to formal verification.

2.2 The Coq Proof Assistant

Coq is an interactive theorem prover based on the Calculus of Inductive Constructions (CIC), a typed lambda calculus that combines a higher-order logic with a richly-typed functional programming language **?**. It provides a formal language to express mathematical assertions and mechanical means to help in the construction of formal proofs.

2.2.1 The Calculus of Inductive Constructions

The Calculus of Inductive Constructions **?** forms the logical foundation of Coq. This type theory extends the Calculus of Constructions with:

- A hierarchy of universes ($\texttt{Type}_0$, $\texttt{Type}_1$, $\texttt{Type}_2$, ...) to avoid paradoxes
- First-class inductive types with constructors and eliminators
- Primitive recursion principles for inductive definitions

In CIC, propositions are represented as types, and proofs are represented as terms (programs) of those types. For example, the proposition "For all natural numbers n, $n = n$" is represented as the type $\forall n : \mathbb{N}, n = n$. A proof of this proposition is a function that takes a natural number n and returns evidence of the equality $n = n$.

```
Definition refl_nat : forall n : nat, n = n :=
  fun n ⇒ eq_refl n.
```

2.2.2 Key Features and Mechanisms

Key features of Coq relevant to our work include:

- **Inductive types**: Allow the definition of data structures like trees through constructors. For example, our rose tree definition:

  ```
  Inductive RoseTree : Type :=
    | Node : nat → list RoseTree → RoseTree.
  ```

- **Dependent types**: Enable the expression of precise specifications and complex properties. For instance, a type that represents lists of exactly length n:

  ```
  Inductive vector (A : Type) : nat → Type :=
    | nil : vector A 0
    | cons : forall n, A → vector A n → vector A (S n).
  ```

- **Tactical language**: Facilitates the construction of complex proofs through composable proof strategies. For example, the `lia` tactic used in our proofs automatically solves linear integer arithmetic goals:

  ```
  Lemma example : forall n : nat, n > 0 → n − 1 < n.
  Proof.
    intros n H. lia.
  Qed.
  ```

- **Extraction mechanism**: Permits the generation of certified executable programs from Coq developments into languages like OCaml or Haskell.

2.2.3 The Curry-Howard Correspondence

The Curry-Howard correspondence—the relationship between computer programs and mathematical proofs—is central to Coq design **?**. Under this paradigm:

- Types correspond to logical propositions
- Programs (terms) correspond to proofs
- Type checking corresponds to proof verification
- Program composition corresponds to proof combination

For example, the logical implication $P \implies Q$ corresponds to the function type $P \to Q$, and a proof of this implication is a function that transforms evidence for P into evidence for Q.

This correspondence allows Coq to serve both as a programming language and as a system for formal mathematical reasoning, making it particularly suitable for formalizing and verifying properties of algorithms and data structures like our hydra game.

2.3 Well-Founded Recursion and Lexicographic Ordering

2.3.1 Well-Founded Relations and Termination

Proving termination of algorithms often relies on identifying a measure that strictly decreases with each recursive call. When this measure is based on a well-founded relation (a relation with no infinite descending chains), termination is guaranteed.

Formally, a binary relation R on a set S is *well-founded* if there are no infinite descending chains $x_1, x_2, x_3, \ldots$ such that $(x_{i+1}, x_i) \in R$ for all $i \geq 1$. In Coq, this is defined as:

```
Inductive Acc {A : Type} (R : A → A → Prop) (x : A) : Prop :=
  | Acc_intro : (forall y, R y x → Acc R y) → Acc R x.

Definition well_founded {A : Type} (R : A → A → Prop) :=
  forall x : A, Acc R x.
```

Well-founded relations are fundamental for defining functions by recursion where the recursive calls may not follow a simple structural pattern. This technique, known as *well-founded recursion*, is essential for proving termination of complex algorithms **?**.

2.3.2 Lexicographic Ordering on Lists

A common approach for complex termination proofs is the use of lexicographic ordering, which compares sequences element by element:

- For sequences $a = [a_1, a_2, ..., a_n]$ and $b = [b_1, b_2, ..., b_m]$
- $a <_{lex} b$ if either:
  - $a_1 < b_1$, or
  - $a_1 = b_1$ and $[a_2, ..., a_n] <_{lex} [b_2, ..., b_m]$

In our Coq formalization, we define this relation inductively:

```
Inductive lex_lt : list nat → list nat → Prop :=
| lex_lt_nil : forall x xs, lex_lt [] (x :: xs)
| lex_lt_cons_lt : forall x y xs ys,
    x < y → lex_lt (x :: xs) (y :: ys)
| lex_lt_cons_eq : forall x xs ys,
    lex_lt xs ys → lex_lt (x :: xs) (x :: ys).
```

This definition captures the three cases of lexicographic comparison:

1. An empty list is lexicographically smaller than any non-empty list
2. If the first elements differ, compare them directly
3. If the first elements are equal, recursively compare the rest of the lists

2.3.3 Example of Lexicographic Ordering

To illustrate lexicographic ordering, consider the following examples:

- $[1, 2, 3] <_{lex} [2, 1, 1]$ because $1 < 2$
- $[3, 1, 4] <_{lex} [3, 2, 1]$ because the first elements are equal ($3 = 3$) and $1 < 2$
- $[5, 6, 7] <_{lex} [5, 6, 7, 8]$ because the shared prefix $[5, 6, 7]$ is equal, and the empty list is less than any non-empty list

2.3.4 Well-Foundedness Properties of Lexicographic Ordering

A crucial property for our termination proof is that lexicographic ordering on lists of natural numbers is well-founded. Intuitively, this means that any descending chain $l_1 >_{lex} l_2 >_{lex} l_3 >_{lex} \ldots$ must eventually terminate.

While the well-foundedness of the standard ordering on natural numbers ($<$) is a primitive notion or a proven property in Coq standard library, proving that its lexicographic extension to lists (`lex_lt`) is also well-founded is a necessary step to fully establish the termination argument from first principles. This proof, while a standard result in termination theory and theoretically trivial given the well-foundedness of the base relation, was not formalized within the scope of the current work. We state it here as a lemma placeholder, acknowledging that its formal proof is required for a complete, self-contained termination argument:

```
Lemma lex_lt_wf : well_founded lex_lt.
```

This lemma states that for any list of natural numbers, there cannot be an infinite descending chain according to our lexicographic ordering. *Our subsequent termination proof relies on the existence and validity of a proof for `lex_lt_wf`.* Fully formalizing the proof of `lex_lt_wf` from the well-foundedness of $<$ on `nat` is a key item for future work and is planned with our tutor for the next research practice.

3 Formalization of the Hydra Game

3.1 Representing Hydras

3.1.1 Rose Trees as Hydras

We represent the hydra as a rose tree, where each node can have an arbitrary number of children:

```
Inductive RoseTree : Type :=
  | Node : nat → list RoseTree → RoseTree.
```

Mathematically, this corresponds to the definition:

$$\text{RoseTree} ::= \text{Node}(n, [t_1, t_2, \ldots, t_k])$$

where $n \in \mathbb{N}$ is a label and $t_1, t_2, \ldots, t_k$ are subtrees.

Each node carries a natural number label and a list of child trees. This structure allows us to represent the hierarchical nature of the hydra, where:

- The root represents the base of the hydra
- Internal nodes represent junctions where multiple branches meet
- Leaf nodes (nodes with empty children lists) represent the hydra's heads

3.1.2 Example Hydra Structures

To illustrate our representation, consider these examples:

1. A simple hydra with one head:

$$\text{Node}(1, [\text{Node}(2, [])])$$

Figure 1: This represents a hydra with a root (labeled 1) and a single head (labeled 2).

2. A hydra with three heads at different depths:

$$\text{Node}(1, [\text{Node}(2, []), \text{Node}(3, [\text{Node}(4, [])])])$$

Figure 2: This represents a hydra with a root (labeled 1) and two branches: one with a single head (labeled 2) and another with an internal node (labeled 3) that has its own head (labeled 4).

3. The binary tree of depth 2 from our formalization:

$$\text{Node}(1, [\text{Node}(2, [\text{Node}(4, []), \text{Node}(5, [])]), \text{Node}(3, [\text{Node}(6, []), \text{Node}(7, [])])])$$

Figure 3: This represents a binary tree of depth 2, with the root labeled 1 and its children labeled 2 and 3, which in turn have their own children.

3.1.3 Counting Nodes by Level

To track the state of the game, we developed a function that counts the number of nodes at each level of the tree:

```
Fixpoint _count_levels (t : RoseTree) : list nat :=
  match t with
  | Node _ children ⇒
      let child_counts := map _count_levels children in
      let merged := fold_left merge_counts child_counts [] in
      1 :: merged
  end.

Definition count_levels (t : RoseTree) : list nat :=
  rev (_count_levels t).
```

Mathematically, this function computes a list $[l_0, l_1, \ldots, l_d]$, where l_i represents the number of nodes at depth i in the tree, and d is the maximum depth. The function works by:

1. Recursively computing the level counts for each child subtree
2. Merging these counts, summing the nodes at corresponding levels
3. Adding 1 for the current node (at level 0 in the subtree)
4. Reversing the list to place the deepest levels first

For example, with the binary tree of depth 2:

$$\text{count\_levels}(\text{binary\_tree\_depth2}) = [4, 2, 1] \tag{1}$$

This indicates 4 nodes at depth 2, 2 nodes at depth 1, and 1 node at depth 0 (the root).

This representation is crucial for our termination proof, as it allows us to track how the hydra's structure changes after each move and show that it decreases according to our lexicographic measure.

3.2 Game Rules

3.2.1 Formal Game Transitions

The game dynamics are formalized through several inductive relations that define how the hydra evolves after each move. These relations operate on the level-count representation of the hydra:

```
Inductive step_internal (n : nat) : list nat → list nat → Prop :=
| StepInternalLong :
    forall prefix x y suffix,
      x > 0 →
      length suffix >= 1 →
      step_internal n (prefix ++x :: y :: suffix)
                      (prefix ++(x − 1) :: (y + n) :: suffix)
| StepInternalTwo :
    forall prefix x y,
      x > 0 →
      step_internal n (prefix ++x :: [y])
```

```
                    (prefix ++(x − 1) :: [y]).

Inductive step_final : list nat → list nat → Prop :=
| StepFinal :
    forall prefix x,
      x > 0 →
      step_final (prefix ++[x]) (prefix ++[x − 1]).
```

Mathematically, these relations define how our list representation of the hydra changes after removing a leaf:

- `StepInternalLong`: For a list $[\ldots, x, y, \ldots]$ where $x > 0$ and there's at least one element after y, the transition produces $[\ldots, x-1, y+n, \ldots]$.
- `StepInternalTwo`: For a list $[\ldots, x, y]$ where $x > 0$ and y is the last element, the transition produces $[\ldots, x-1, y]$.
- `StepFinal`: For a list $[\ldots, x]$ where $x > 0$ and x is the last element, the transition produces $[\ldots, x-1]$.

These relations capture the two main cases in our game:

- `step_internal`: When removing a leaf causes new heads to grow from its grandparent
- `step_final`: When removing a leaf connected to the root, which doesn't trigger new growth

3.2.2 Unified Step Relation

We combine these cases into a unified `step` relation that represents a single move in the game:

```
Inductive step (n : nat) : list nat → list nat → Prop :=
| Step_internal_case :
    forall l1 l2,
      step_internal n l1 l2 →
      step n l1 l2
| Step_final_case :
    forall l1 l2,
      step_final l1 l2 →
      step n l1 l2.
```

This relation provides a complete specification of the game dynamics: given any hydra state (represented as a list of node counts), it defines all possible next states.

3.2.3 Termination Conditions

Additionally, we define a `step_done` relation that identifies when the game has terminated:

```
Inductive step_done : list nat → Prop :=
| StepDoneEmpty :
    step_done []
| StepDoneNonEmpty :
    forall l,
      l <> [] →
      (forall x, In x l → x = 0) →
      step_done l.
```

This relation captures two termination conditions:

1. The hydra is completely empty (represented by an empty list)
2. The hydra consists only of nodes with no leaves (represented as a non-empty list containing only zeros)

3.2.4 Example Game Sequence

To illustrate how these relations model the game, consider the following sequence of steps from our formalization (with growth factor $n = 2$):

$$[4, 2, 1] \xrightarrow{\text{step}} [3, 4, 1] \tag{2}$$

$$[3, 4, 1] \xrightarrow{\text{step}} [2, 6, 1] \tag{3}$$

$$[2, 6, 1] \xrightarrow{\text{step}} [1, 8, 1] \tag{4}$$

$$[1, 8, 1] \xrightarrow{\text{step}} [0, 10, 1] \tag{5}$$

$$[0, 10, 1] \xrightarrow{\text{step}} [0, 9, 1] \tag{6}$$

$$[0, 9, 1] \xrightarrow{\text{step}} [0, 8, 1] \tag{7}$$

In this sequence:

- The first four steps remove leaves at the deepest level (level 0 in the reversed representation), causing growth at level 1
- Once level 0 has no more leaves (count = 0), the game proceeds to remove leaves at the next level
- The game continues until all levels except the root level have count 0

This example demonstrates how our formalization captures the game dynamics and the progressive reduction of the hydra.

4 Termination Proof

Our termination proof consists of several key lemmas and theorems that together establish that each game step leads to a lexicographically smaller state.

4.1 Preservation of Length

First, we prove that game steps preserve the length of our representation lists:

```
Lemma step_internal_preserves_length :
  forall n l1 l2,
    step_internal n l1 l2 → length l1 = length l2.

Lemma step_final_preserves_length :
```

```
  forall l1 l2,
    step_final l1 l2 → length l1 = length l2.

Theorem step_preserves_length :
  forall n l1 l2,
    step n l1 l2 → length l1 = length l2.
```

Mathematically, these lemmas establish that:

$$\forall n \in \mathbb{N}, \forall l_1, l_2 \in \text{List}(\mathbb{N}), \text{step}(n, l_1, l_2) \implies |l_1| = |l_2|$$

These lemmas establish that when we transition from one game state to another, the length of our representation (corresponding to the maximum depth of the hydra) remains unchanged. This is important because:

- It ensures that our lexicographic comparison is well-defined, as we're comparing lists of the same length
- It confirms that the game doesn't add new levels to the hydra, which aligns with the game rules
- It demonstrates that the growth happens within existing levels, not by adding new ones

4.2 Decreasing Lexicographic Measure

The core of our termination proof consists of showing that each game step produces a lexicographically smaller representation:

```
Lemma step_internal_decreases_lex :
  forall n l1 l2,
    step_internal n l1 l2 →
    lex_lt l2 l1.

Lemma step_final_decreases_lex :
  forall l1 l2,
    step_final l1 l2 →
    lex_lt l2 l1.

Theorem step_decreases_measure :
  forall n l1 l2,
    step n l1 l2 →
    lex_lt l2 l1.
```

Mathematically, the main theorem states:

$$\forall n \in \mathbb{N}, \forall l_1, l_2 \in \text{List}(\mathbb{N}), \text{step}(n, l_1, l_2) \implies l_2 <_{lex} l_1$$

The proof of `step_internal_decreases_lex` handles two cases:

1. When the hydra configuration has the form prefix$+[x, y]+$suffix with $x > 0$ and a non-empty suffix:

```
induction prefix; simpl.
+ (* prefix = [] *)
  apply lex_lt_cons_lt.
  lia. (* x - 1 < x since x > 0 *)
+ (* prefix = a :: prefix' *)
  apply lex_lt_cons_eq.
  apply IHprefix.
```

This case demonstrates that removing a leaf at a deep level (reducing x by 1) and adding new branches at a higher level (increasing y by n) still results in a lexicographically smaller configuration.

2. When the hydra configuration has the form prefix $+ [x, y]$ with $x > 0$ and no suffix:

```
induction prefix; simpl.
+ (* prefix = [] *)
  apply lex_lt_cons_lt.
  lia. (* x - 1 < x since x > 0 *)
+ (* prefix = a :: prefix' *)
  apply lex_lt_cons_eq.
  apply IHprefix.
```

This handles the case of removing a leaf at the second-to-last level, with similar reasoning.

Similarly, `step_final_decreases_lex` shows that removing a leaf at the root level always decreases the lexicographic measure.

These theorems establish that:

- Internal steps (where removing a leaf triggers growth) decrease the lexicographic measure
- Final steps (where removing a leaf doesn't trigger growth) also decrease the measure
- As a result, any valid game step decreases the lexicographic measure

The key insight is that while the hydra may grow in size (total number of nodes), the distribution of nodes across levels changes in a way that always produces a lexicographically smaller configuration.

4.3 Termination Argument

Our `step_decreases_measure` theorem (Section **??**) establishes that each game step produces a configuration that is lexicographically smaller than the previous one according to the `lex_lt` relation:

$$\forall n \in \mathbb{N}, \forall l_1, l_2 \in \text{List}(\mathbb{N}), \text{step}(n, l_1, l_2) \implies l_2 <_{lex} l_1$$

Given this property, and leveraging the fact that the lexicographic ordering `lex_lt` on lists of natural numbers **is** a well-founded relation (as stated by `lex_lt_wf`, though its formal proof is not included in this specific formalization as discussed in Section **??**), we can conclude that:

- There cannot be an infinite sequence of game steps $l_0, l_1, l_2, \ldots$ where $\text{step}(n, l_i, l_{i+1})$ for all i.
- Therefore, the game must terminate after a finite number of moves, reaching a state satisfying the `step_done` condition.

More formally, the termination theorem states:

Theorem 1 (Hydra Game Termination) *For any initial hydra configuration $l_0 \in List(\mathbb{N})$ and growth factor $n \in \mathbb{N}$, there exists a finite sequence of configurations $l_0, l_1, l_2, \ldots, l_k$ such that:*

1. *For all $0 \leq i < k$, $step(n, l_i, l_{i+1})$ holds.*
2. *$step\_done(l_k)$ holds.*

The proof of this theorem relies directly on two key components: first, the established property $\text{step}(n, l_i, l_{i+1}) \implies l_{i+1} <_{lex} l_i$, demonstrating that each step reduces the measure, and second, the well-foundedness of the *lex_lt* relation itself. Since a well-founded relation has no infinite descending chains, a process that strictly decreases according to such a relation must be finite.

This result is significant because it establishes, through formal verification (conditional on the well-foundedness of the measure), that the simplified Hydra Game always terminates regardless of the initial configuration. The proof is constructive in nature, providing a concrete measure that strictly decreases with each step.

4.4 Connections to Ordinal Theory

The termination of the Hydra Game is closely related to ordinal theory. While our simplified formalization uses lexicographic ordering on lists of natural numbers, the full Kirby–Paris result involves transfinite ordinals below ε_0 (the first fixed point of the ordinal exponentiation function) **?**.

In the original game, each configuration corresponds to an ordinal, and each step reduces this ordinal according to certain rules. The termination proof then relies on the well-foundedness of ordinals below ε_0. Specifically, an ordinal assignment function maps each hydra to an ordinal in such a way that:

- Each valid move decreases the ordinal
- The minimal ordinal (0) corresponds to the empty hydra

For the original Kirby-Paris game with unrestricted growth, the appropriate ordinals are much larger than those needed for our simplified variant. The ordinal assignment becomes more complex, involving ordinal arithmetic operations like addition, multiplication, and exponentiation **?**.

The fact that the termination of the full Hydra Game cannot be proven within Peano arithmetic is related to the fact that the well-foundedness of ordinals below ε_0 cannot be established within this system. This connects our work to fundamental results in proof theory and the foundations of mathematics **?**.

5 Discussion and Related Work

Our formalization demonstrates a simplified approach to verifying the termination of the Hydra Game. While this approach does not capture the full complexity of the original Kirby–Paris result, it provides a concrete, accessible example of formal verification using the Coq proof assistant.

The Hydra Game belongs to a family of mathematical results that demonstrate the limitations of formal systems, including:

- Goodstein's theorem **?**, which also exhibits behavior that cannot be proven to terminate within Peano arithmetic
- The Paris–Harrington theorem, another natural mathematical statement independent of Peano arithmetic
- Various termination problems in term rewriting systems **?**, which often require sophisticated ordinal-based techniques

Our approach using lexicographic ordering relates to broader work on termination proofs in computer science, particularly in:

- Program verification, where lexicographic ordering is a standard technique for proving termination of complex algorithms
- Term rewriting theory **?**, which uses similar well-founded orderings to establish termination properties
- Interactive theorem proving **?**, where such techniques are formalized and mechanically verified

6 Conclusion and Future Work

In this paper, we presented a formalization of a simplified variant of the Kirby–Paris Hydra Game using the Coq proof assistant. Our formalization:

- Represents hydras as rose trees and game states as lists of node counts.
- Defines game rules as inductive relations on these representations.
- Proves that each game step strictly decreases a measure based on the lexicographic ordering of node counts.
- Establishes termination by leveraging the well-foundedness of the lexicographic ordering on lists of natural numbers. (Note: while crucial for the proof, the formal well-foundedness of the lexicographic ordering itself was assumed as a standard result and not proven within this specific Coq development, as detailed in Section **??**).
- Provides visualizations that build intuition about the game dynamics.

Our work demonstrates the application of formal methods to verify properties of combinatorial games with connections to fundamental results in mathematical logic. It also serves as an accessible case study in formal verification using the Coq proof assistant.

Future work could extend this formalization in several directions:

- **Formalizing the well-foundedness of lexicographic ordering:** A key missing piece in the current formalization is the formal proof that the `lex_lt` relation on lists of natural numbers is well-founded, building upon the well-foundedness of the standard `nat` ordering. This formalization step is a planned activity for our next research practice, conducted with our tutor.
- Formalizing the full Kirby–Paris game with its more complex growth rules.

- Connecting the formalization to ordinal theory to establish the relationship with Peano arithmetic.
- Extracting certified algorithms for playing and analyzing the game.
- Developing more sophisticated visualizations of the proof structure.

The Hydra Game continues to be a fascinating object of study at the intersection of combinatorial game theory, proof theory, and formal verification. Our formalization contributes to making these connections more explicit and accessible through mechanized proof.

References

Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

Wilfried Buchholz. An independence result for $(\Pi^1_1 - CA) + \mathrm{BI}$. *Annals of Pure and Applied Logic*, 33:131–155, 1987.

Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic*, pages 50–66. Springer, 1988.

Nachum Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 3(1-2):69–115, 1987.

Jean Gallier. Constructive logics part I: A tutorial on proof systems and typed -calculi. *Theoretical Computer Science*, 110(2):249–339, 1993.

Reuben Louis Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, 9(2):33–41, 1944.

Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14(4):285–293, 1982.

Lawrence C Paulson. A framework for defining logics. *Journal of the ACM (JACM)*, 36(3):479–508, 1989.

Michael Rathjen. *Well-ordering principles in proof theory and reverse mathematics*. Springer, 2017.