A Rose Tree Is Blooming (Proof Pearl)

Joomy Korkut
Bloomberg
New York, USA
jkorkut@bloomberg.net

Abstract

Game trees are a fundamental mathematical abstraction for analyzing games, often implemented as rose trees in functional programming. We can construct rose trees by starting from an initial state and iteratively applying the function that provides the states one move away, an approach known as an *anamorphism* (or colloquially, an *unfold*).

In ML or Haskell, finite and infinite rose trees share the same type, but proof assistants typically distinguish them: finite trees as inductive, infinite ones as coinductive. With the inductive approach, defining the unfold function is tricky but we obtain a finite tree that is easy to compute with and easy to reason about. With the coinductive approach, defining the unfold function is easy but we obtain a potentially infinite tree, which is harder to reason about since we must resort to proof techniques like bisimulation. In this paper, we compare the inductive and coinductive approaches to game trees, implement unfold functions for both, and prove the soundness and completeness of both unfold functions (with respect to the game) in the Rocq Prover. We illustrate these techniques with implementations of a tic-tac-toe game and a simple SAT solver.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Functional languages**; • **Computing methodologies** → Game tree search.

Keywords: formal verification, termination, recursion, corecursion, game trees, Rocq

ACM Reference Format:
Joomy Korkut. 2026. A Rose Tree Is Blooming (Proof Pearl). In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779031.3779091

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779091

1 Introduction

Game trees are a useful mathematical abstraction for reasoning about pathfinding, constraint satisfaction, and games [1]. A game tree consists of the states of a game organized as a tree, where each branch leads to a state after a move. A complete game tree is one that contains all possible game states. Such trees can be used for global reasoning about future moves, such as deterministically computing the best possible move at a given point in the game.

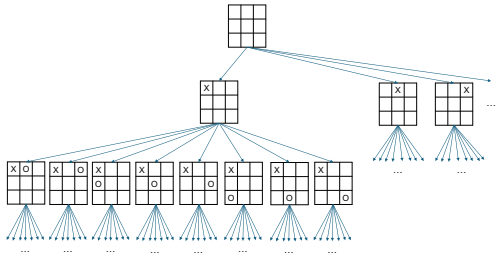

Figure 1. An example game tree for tic-tac-toe.

An example game tree for tic-tac-toe can be seen in Figure 1. At the root node, we start with the empty board, which is the initial game state. First it is **x**'s turn, so we consider all possible moves **x** can make. There are 9 empty cells on the board **x** could choose, so the root node has 9 immediate subnodes. At each of the subnodes, it is now **o**'s turn. There are 8 empty cells left on the board at any of these nodes, therefore they all should have 8 immediate subnodes.

We omit most branches of the tic-tac-toe game tree here for brevity. The complete tree would have 549,946 nodes if we expand naively, and that is if we were to stop building the game tree when a player wins before all cells are filled. The lesson to learn here is that game trees can grow exponentially, and that computing game trees is hard! For games with a large number of possible game states, it may even be infeasible. Famously, the chess games of IBM's Deep Blue against grandmaster and then long-time world champion Garry Kasparov exemplified the difficulty of computing game trees and searching for the best move [27, 46].

The straightforward implementation of game trees as a functional data structure coincides with what the functional programming community calls *rose trees*: trees where nodes can have a list of subtrees. This type has been the subject of considerable research, both in the functional programming community [35, 21, 23, 22, 6, 25], due to its challenges

for program calculation, and in the proof assistants community [45, 60, 63, 54], due to the trickiness of code generation for it. (As they say, there is no rose without a thorn!) That is, however, not what we want to focus on in this paper. Instead, we want to investigate how the restrictions of proof assistants can complicate building and evaluating game trees represented as rose trees. For this purpose, we will use[1] the Rocq Prover (formerly known as Coq) [49], which is a purely functional programming language and a proof assistant.

The most intuitive way of populating a game tree is to start from an initial game state and to keep adding the states that are one move away from a given state. Suppose we have a function that takes a single game state as an input, and returns a list of game states, each representing the new game state after one move. We should now be able to write a function that computes the entire game tree from these. Luckily, functions that build data structures by applying a function repeatedly have been studied extensively [36]; these functions are called *anamorphisms* (or *unfolds*)! Unluckily, anamorphisms yield potentially infinite values, and therefore their results are often written as *coinductive* values, such as streams [64, 37]. On the other hand, defining anamorphisms to generate *inductive* values is awkward, since it requires nonstructural recursion, which is difficult in Rocq because of termination restrictions [48, 7]. Therefore, we have to think carefully about whether we should define our rose trees inductively, which guarantees that our rose trees would be finite, or coinductively, which allows infinite rose trees.

This decision requires us to figure out a more important issue: Is the game under consideration even finite? For tic-tac-toe, we can see that the game will end once all the cells are filled or a player wins the game, but what about a game like chess? Would a complete game tree for chess be finite?[2] If our game is finite, then we might want to use inductive rose trees to represent its game tree, which makes building the game tree more difficult but makes proving properties about the game tree easier. If our game is infinite, we might want to use coinductive rose trees, since coinductive values allow representing infinite game trees. Coinductive trees would make building the game tree much easier but reasoning about the entire tree harder. In this paper, we *implement* game trees both inductively and coinductively, and investigate the advantages and disadvantages of these approaches in Rocq. Concretely, we

1. define coinductive rose trees and a corecursive unfold `unfold_cotree` (§3), and show how to recover finiteness via a predicate `finite_cotree` over a fuel-led conversion,
2. define an inductive unfold `unfold_tree` driven by accessibility predicates, with an intrinsic interface for `next`, made structurally recursive (§4),
3. prove *soundness* and *completeness* theorems for both unfoldings (§5.1, §5.2),
4. showcase these definitions and functions on tic-tac-toe, whose termination we prove through a well-founded game step relation, and extract an executable game against an unbeatable AI (§6),
5. and repurpose the same framework to build a simple SAT solver, expressed as a game tree that enumerates assignments and returns a satisfying one if it exists (§7).

2 A naive attempt

For our initial approach, let us use the following inductive definition of rose trees:

```
Inductive tree (A : Type) : Type :=
| node : A → list (tree A) → tree A.
```

We have an inductive type `tree`, parameterized by the type `A`, which represents the data stored in the nodes of the rose tree. We have a single constructor `node`, which can hold any number of subtrees with the help of the `list` type.

Now, we can start writing the function that builds a game tree. Our initial game state will be the root of the game tree, and we can call `next` to get the states one move away.

If the `next` function returns an empty list, we stop expanding that branch, since there are no more moves to make. If the game is finite, all branches of the game tree will eventually reach a state at which there are no more moves to make. If the `next` function returns some states, these will be the roots of the subtrees. Therefore, we can do a recursive call for each of them and continue building the subtrees.

Let us try to implement this function, generalized to any game state type `A`:

```
Fixpoint unfold_tree {A : Type}
                     (next : A → list A)
                     (init : A) : tree A :=
  node init (map (unfold_tree next) (next init)).
```

```
Error:
Cannot guess decreasing argument of fix.
```

Our attempt here failed because we could not convince Rocq that our function terminates. Non-terminating functions make a proof assistant unsound (since you could prove every theorem, including `False`, with an infinite loop), therefore Rocq only accepts *obviously terminating* functions[3]. An obviously terminating recursive function is usually[4] one

[1] The Rocq development for this paper is available as supplementary material at https://github.com/bloomberg/game-trees.

[2] Actually, yes, since the 75 move rule was introduced [38].

[3] Since Rocq cannot detect whether any given program terminates or not (also known as the *halting problem*, whose undecidability was famously proven by Turing [61]), Rocq chooses to accept a subset of programs that terminate.

[4] Rocq's termination checker accepts some special cases that are not strictly structurally recursive but are terminating. For the full formalization of the termination checker, see Tan [59].

that is structurally recursive, where one of the arguments passed to the recursive call is a strict subterm of the original argument.

In our definition of `unfold_tree`, however, the recursive call is not given a strict subterm of the argument; it is given an element drawn from the result of a call to `next`. The argument in the recursive call may even be larger than the original argument! We are at the mercy of whoever implemented and passed the `next` function as an argument. How do we deal with this situation? We need nonstructural recursion to implement this function, but even then we have to show that the function eventually terminates.

The question we want to ask ourselves here is, is our game finite? If the game is not finite, then Rocq is right to disallow this particular function definition, and we should turn to coinductive rose trees. If the game is finite, how do we convince Rocq that our function terminates? Or can we pretend initially that our tree might be infinite and later prove that it actually terminates? We pursue both paths: We will first investigate the coinductive trees, and later see how we can finish the `unfold_tree` function for inductive trees.

3 Coinductive types to the rescue!

Our earlier inductive definition of `tree`s limits the trees we can create to finite ones. Rocq, however, also allows possibly infinite values using coinductive types. Let us define a coinductive type for potentially infinite rose trees:

```
CoInductive cotree (A : Type) : Type :=
| conode : A → colist (cotree A) → cotree A.
```

Notice how we used `colist`s here, which are the coinductive version of the inductive type `list`. While a `colist` can consist of infinite applications of the `cocons` constructor, a `colist` can also be terminated with the `conil` constructor.

Using infinite rose trees, we can implement the same functions as we did before. In fact, even our naive `unfold_tree` definition, when adjusted to use `cotree`s, is accepted by Rocq as a valid corecursive function:

```
CoFixpoint unfold_cotree
           {A : Type}
           (next : A → colist A)
           (init : A) : cotree A :=
  conode init (comap (unfold_cotree next) (next init)).
```

3.1 Proving the termination of cotrees

`cotree`s can be infinite or finite. They can be infinite because Rocq allows corecursive definitions as long as the recursive calls are arguments to a constructor of the coinductive type. This means a definition of an infinite tree like this is allowed:

```
CoFixpoint infinite_tree (n : nat) : cotree nat :=
  conode n (cocons (infinite_tree (1 + 2 * n))
             (cocons (infinite_tree (2 + 2 * n) conil)).
```

This infinite tree definition allows building trees like this:

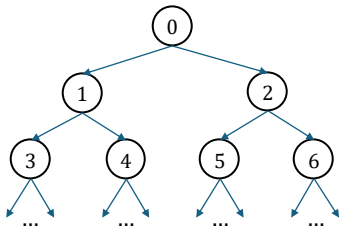

Figure 2. `infinite_tree 0` visualized.

But coinductive trees can also be finite. A `colist` of `cotree`s can be terminated by a `conil`, therefore it is possible to construct a finite `cotree` such as `conode 0 conil`.

One way we can inspect whether a `cotree` is finite is to unwrap it into an inductive `tree`. However, to make sure that we unwrap the `cotree` a finite number of times, we add an extra parameter `fuel`, which is a natural number, to our function. This is a common pattern in proof assistants, called the *fuel pattern* [34, 57]. Otherwise, the unwrapping is straightforward: we inspect the `cotree` and put the label onto a `tree`, and continue with recursion on the `fuel`. Following the OCaml tradition of conversion function names, we name the unwrapping function for lists `list_of_colist` and the one for trees `tree_of_cotree`:

```
Fixpoint tree_of_cotree
         {A : Type} (fuel : nat)
         (t : cotree A) {struct fuel} : tree A :=
  match t with
  | conode a f ⇒
    match fuel with
    | O ⇒ node a nil
    | S fuel' ⇒
      node a (map (tree_of_cotree fuel')
                  (list_of_colist fuel f))
    end
  end.
```

It is now possible to reach a complete inductive game tree from a `cotree` by unwrapping the tree sufficiently many times. Here is what this termination condition looks like in Rocq as a predicate on `cotree`s:

```
Definition finite_cotree
           {A : Type} (t : cotree A) : Type :=
  {n : nat |   tree_of_cotree n t
             = tree_of_cotree (1 + n) t}.
```

This definition says that a cotree can be called finite if there exists a natural number `n` such that unwrapping the cotree `n` times results in the exact same tree as unwrapping the cotree `1 + n` times. Intuitively, there is an `n` that is enough fuel to capture the whole tree.

We can even define a predicate on an initial game state and the function that gives us the game states after one move:

```
Definition finite_game
           {A : Type}
           (next : A → colist A)
           (init : A) : Type :=
  finite_cotree (unfold_cotree next init).
```

Using these predicates, we can prove that the game tree for tic-tac-toe is complete, even though checking this proof can take around half a minute:

```
Theorem ttt_is_finite : finite_game ttt_conext ttt_init.
Proof. exists 10; simpl; reflexivity. Defined.
```

Why did we have to unwrap the tree 10 times? Well, the depth of the game tree for tic-tac-toe is 10: The initial state accounts for one, and a tic-tac-toe game must end within 9 moves, so the total accounts for all 10 steps of depth. While we can find the game tree depth (and the maximum number of moves) intuitively for tic-tac-toe, finding this number for more complex games is not this straightforward. For example, it took a lot of computational power and clever tricks to find that the maximum number of moves in chess is 17,697 [32, 38]. For games like this, we may want to compute the game tree inductively, where we do not have to know the maximum number of moves in a finite game.

4 Finding a way around termination

Rocq provides multiple mechanisms to write functions with nonstructural recursion: the `Fix` combinator from the Rocq standard library, the **`Function`** command [3, 4], the **`Program Fixpoint`** command [52], the Equations plugin [55], or just *accessibility predicates* [42, 7] in plain Rocq. The same idea of well-founded recursion lies at the heart of all of these methods[5]: that *something* keeps getting "smaller" with every recursive call, and there should be a point at which it cannot get any smaller. To say it more precisely, if the argument to the recursive function decreases with respect to a relation, and if the relation does not allow infinite descending chains, then the function eventually has to terminate.

Accessibility predicates are the constructive formulation of the same idea. They are defined in the Rocq standard library as follows:

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

We can think of the accessibility predicate as a recursion permission [31]. For a relation `R`, if we have the permission for `x`, that means we can obtain the permission for any value `y` that is "less than" `x` (as judged by the relation `R`). Since the new `Acc` value is obtained from a strict subterm of the original, this counts as structural recursion! We will, however, have to prove that the relation is *well-founded*: that we can produce an `Acc` for any `x` of the type `A`.

A common pattern in using accessibility predicates for nonstructural recursion is having two functions:

1. an auxiliary function that is structurally recursive on its `Acc` parameter,
2. an entry point function that calls the auxiliary function with the necessary `Acc` argument, produced by a well-foundedness proof for the relation `R`.

For cases in which the entry point function does not use a fixed relation, we can quantify over the well-foundedness of the relation using a type class [56]:

```
Class WellFounded {A : Type} (R : A → A → Prop) :=
  wellfounded : ∀ (x : A), Acc R x.
```

When we pick a relation to use for termination, we create an instance of the `WellFounded` type class for that relation by proving its well-foundedness. Our entry point function that quantifies over the relation can also have a `WellFounded` type class parameter and use that parameter to obtain the accessibility predicate.

4.1 Unfolding inductive trees

Let us get back to our definition of `unfold_tree`. When we make recursive calls on the next states, what determines that progress is made? To ask more colloquially, what gets smaller in our recursive calls? We are deeper in the game, but does this mean the game state is somehow "smaller" in a way we can express to Rocq's type checker? For tic-tac-toe, it is trivial to see that the number of empty cells gets less as the game continues. But what about other games? For a game of chess, for example, how can we say a game state is closer to the end than another game state? We leave this question to a brave soul who wants to take on the task of writing a Rocq proof of the finiteness of chess. For now, we parametrize our `unfold_tree` function with the `WellFounded` type class and move on.

Our new plan is to write an auxiliary function and an entry point function. Let us start with the auxiliary function, which will do the heavy lifting. Compared to our naive attempt at `unfold_tree`, our new auxiliary function must have new parameters: `R`, which is a relation on the game state type `A`, and `acc : Acc R init`, the accessibility predicate that allows us to do a recursive call on anything of type `A` that is "smaller" than `init`. But how do we know that the values we obtain from the `next` function are "smaller" than `init`? We do not. We have to find a way to make it so. If we change the type of `next` to carry a proof that all of the elements in the `list` are "smaller" than `init`, that solves our problem.

The style of dependently typed programming where functions return a result and a proof about the result at the same

[5] The mechanisms we listed above amount to syntactic sugar that generate the same obligations. Working directly with `Acc` highlights how termination evidence is threaded through our definitions, which fits the didactic spirit of a proof pearl. Also, as Leroy [33] argues, functions using `Acc` are often easier to write, prove properties about, and extract.

time is called *intrinsic* verification. This style is often contrasted with *extrinsic* verification, where you write your function in an ML-like subset of your language, and then prove properties about it outside of the function definition [31]. Extrinsic verification is often favored over intrinsic verification in Rocq[6]. However, in this case, we are forced to take a more intrinsic approach since we need the extra proof to convince Rocq that our function terminates.

Now, with the extra parameters and the updated type for `next`, we can write down the type of our auxiliary function (but let us leave it unimplemented for now!) and the implementation of our entry point function:

```
Fixpoint unfold_tree_aux
         {A : Type} (R : A → A → Prop)
         (next : ∀ (y : A),
                 {l : list A | Forall (λ x ⇒ R x y) l})
         (init : A)
         (acc : Acc R init) {struct acc} : tree A :=
  (* We have to fill here! *).

Fixpoint unfold_tree
         {A : Type}
         (R : A → A → Prop) `{WellFounded R}
         (next : ∀ (y : A),
                 {l : list A | Forall (λ x ⇒ R x y) l})
         (init : A) : tree A :=
  unfold_tree_aux R next init (wellfounded init).
```

We want to fill in the definition of `unfold_tree_aux`, but we would like a concise definition, ideally one as close as possible to our naive attempt. Earlier, we had called `next` with the initial state `init`, and then tried to `map` the result by converting each of the states after one move into their own game trees. Would it be possible to follow the same idea?

4.1.1 A short detour with lists and dependent pairs.

The new type for `next` makes it more difficult to mimic our naive implementation of `unfold_tree`, as the new return type of `next` is a `list` paired with a proof that all elements in the list are "less" than `init`. [7] The `map` function, however, expects a `list` and not a dependent pair. The left side of the pair contains a `list`, but we cannot simply get the first projection of the pair and keep mapping, because we need the proof parts to be able to do recursive calls! That is, we need a function that takes the dependent pair of the `list` and a proof about all elements in that list, and divides it into a list containing elements paired with a proof about each particular element. Let us try to define this function recursively:[8]

```
Fixpoint distribute
         {A : Type} {P : A → Prop}
         (p : {l : list A | Forall P l})
         : list {a : A | P a} :=
  match p with
  | ([]; _) ⇒ []
  | (x :: xs; Forall_cons pf_x pf_xs) ⇒
    (x; pf_x) :: distribute (xs; pf_xs)
  end.
```

```
Error:[9]
- Non-exhaustive pattern-matching:
  no clause for pattern (_::_; Forall_nil).
- Recursive call has argument (xs; pf_xs)
  instead of a subterm of p.
```

Our attempt fails for two reasons: the first is that Rocq is not particularly good at dependent pattern matching. Even though the `::` case of the **`match`** expression correctly assumes that the proof packed in the dependent pair must be talking about a nonempty list, Rocq could not understand that. While there are tricks like the *convoy pattern* [9] to write a **`match`** expression that Rocq understands, using tactics that eventually generate that possibly complicated **`match`** expression for us might be a good idea. Rocq tactics are usually for writing proofs, but there is no reason we cannot use them for generating program terms instead of proof terms!

The second reason we failed is that `distribute` is not structurally recursive on its input `p`, in the way Rocq would expect. The argument to the recursive call is not a direct subterm of the original input; the argument is a new dependent pair that consists of subterms of the original input. This is not acceptable to the termination checker. But wait a minute! If this function has to unpack a dependent pair to get its content, only to put subterms of the content in a dependent pair again for the recursive call, why do we need a dependent pair at all? We do not, so let us write a helper function that avoids that indirection by splitting the dependent pair parameter into two parameters.

Here is the helper function we need, that uses the strategies we decided on: tactics for dependent pattern matching, and splitting the dependent pair parameter into two:

[6] Mostly due to the user experience of dependent pattern matching, as we can see in subsubsection 4.1.1. While Rocq technically supports dependent pattern matching, writing such code often requires the use of convoy pattern [9] or generating this code with Ltac tactics, although writing Rocq code with Agda-like dependent pattern matching is possible thanks to the Equations [55] plugin in Rocq.

[7] You might wonder why we use `{l : list A | ...}` instead of `list {a : A | ...}` here. The choice reflects a trade-off: With `list {a : A | ...}`, it's easier to implement unfolding, but writing an intrinsically terminating next moves function becomes more difficult. With `{l : list A | ...}`, the situation is reversed: unfolding is harder, but defining the next moves function is easier. Since our priority is to make the library easier to use, we adopt the second approach. This is illustrated by our definition of `ttt_next_intrinsic` in section 6.

[8] `(_; _)`, `_.1`, and `_.2` are custom notations we defined for construction, first projection, and second projection of `sig`. Rocq standard library already uses these notations for the same purposes for the very similar `sigT` type. We use `sig` instead of `sigT` for better erasure in program extraction.

[9] This speech bubble is not the actual error message for this program. The first one is closer to the actual one; the second error message is what you get if you handle the dependent pattern matching and specify which parameter the function is structurally recursive on.

```
Fixpoint zip_proofs
         {A : Type} {P : A → Prop}
         (l : list A) (pf : Forall P l) {struct l}
         : list {a : A | P a}.
Proof.
  destruct l as [| x xs ].
  - refine [].
  - refine ((x; _) :: @zip_proofs A P xs _);
    inversion pf; auto.
Defined.
```

Notice how we use the **refine** tactic to leave holes (_) in the result we are returning, only to fill them with the subproofs we get from the **inversion** of pf.

Now we are able to define our distribute function as a simple call to zip_proofs:

```
Definition distribute
           {A : Type} {P : A → Prop}
           (p : {l : list A | Forall P l})
           : list {a : A | P a} :=
  zip_proofs p.1 p.2.
```

4.1.2 Back on track with unfolding.

Now we have all the necessary machinery to finish defining unfold_tree_aux, whose complete definition is as follows:

```
Fixpoint unfold_tree_aux
         {A : Type} (R : A → A → Prop)
         (next : ∀ (y : A),
                 {l : list A | Forall (λ x ⇒ R x y) l})
         (init : A)
         (acc : Acc R init) {struct acc} : tree A :=
  match acc with
  | Acc_intro acc' ⇒
    node init (map (fun '(x; pf) ⇒
                      unfold_tree_aux R next x
                                      (acc' x pf))
              (distribute (next init)))
  end.
```

When next is called, the dependent pair (of the list and the proof that all elements of the list are "less" than init) is then distributed into a list of dependent pairs that consist of a single element and a proof of that particular element being "less" than init. Now we have something we can map! Since the list we got from distribute contains the states after one move, we can map this list into the subtrees obtained from recursive calls, which we can do in a provably terminating way since we carry pf, the proof that x is indeed "less" than init.

We had already defined unfold_tree above, therefore this concludes the definition of unfolding for inductive trees.

> Our unfold_tree function requires explicitly choosing an initial state init and a next-states function next.

> Could we instead derive these from a predefined, well-founded relation on game states? Not automatically. We might not even know if our game is *finitely branching*. Consider a game where two players each pick a positive integer, and the larger number wins. The complete tree depth is 3 (initial state, player 1's move, player 2's move), but each player has infinitely many moves, making the game horizontally infinite. We cannot completely express this game using the next function that returns a list (or a dependent pair containing a list), since list is an inductive type and can only express a finite number of possible next moves. But if we use coinductive game trees and unfold_cotree, where our next function returns a colist, we can express this game.

5 Soundness and completeness

The unfold_tree and unfold_cotree functions, solely by looking at their types, are guaranteed to return a finite tree in the inductive case, and a possibly infinite (but perhaps still finite) tree in the coinductive case. There is, however, no theorem that tells us what is in this tree. Without inspecting the implementation of these two functions, we cannot tell if our implementation is faithful to the initial state and next move function we provided. More specifically, we want to prove two properties:

1. the soundness of unfolding, i.e., if there is a game state in the unfolded tree, there must be a sequence of moves from the initial state to that state,
2. the completeness of unfolding, i.e., if there is a sequence of moves from the initial state to a state, that state must be in the unfolded tree.

5.1 Reasoning inductively

Let us start with inductive trees. To be able to reason about the correctness of the unfold_tree function, we need to develop a vocabulary of predicates. The first notion we want to express in Rocq is what it means to be *in a tree*. The In relation from the Rocq standard library already defines what it means to be in a list. We similarly define a relation In_tree for trees:

```
Inductive In_tree
          {A : Type} (a : A) : tree A → Prop :=
| In_this : ∀ f, In_tree a (node a f)
| In_that : ∀ a' f,
    Exists (In_tree a) f → In_tree a (node a' f).
```

If a game state a is in a tree, a is either the label of the root node, or there is[10] a tree among the children f in which a occurs.

[10] Exists is an inductive type from the Rocq standard library. Exists (In_tree a) f is equivalent to ∃ t, In t f ∧ In_tree a t.

The next definition we need in our vocabulary is one that expresses what it means for one game state to follow another. We can call this a `step` in the game:

```
Definition step
    {A : Type} {R : A → A → Prop}
    (next : ∀ (a : A),
        {l : list A | Forall (λ x ⇒ R x a) l})
    (x y : A) : Prop :=
  In y (next x).1.
```

The `next` function returns a pair of a list of next moves, and a proof that all of those moves are "less" than the previous move according to the relation `R`. A relation that asserts that a state is `In` the first projection of a call to `next` expresses what it means to be a move away from a game state.

Finally, we can define a relation of reachability from one game state to another game state:

```
Definition reachable
    {A : Type} {R : A → A → Prop}
    (next : ∀ (a : A),
            {l : list A | Forall (λ x ⇒ R x a) l})
   : A → A → Prop :=
  clos_refl_trans A (step next).
```

The reflexive-transitive closure of `step` expresses the reachability of one state from another in a finite number of moves.

Now we have the necessary vocabulary to express the soundness and completeness of unfolding for inductive trees:

```
Theorem unfold_tree_sound :
  ∀ {A : Type} (R : A → A → Prop) `{WellFounded R}
    (next : ∀ (a : A),
            {l : list A | Forall (λ x ⇒ R x a) l })
    (init : A),
  ∀ (a : A),
    In_tree a (unfold_tree R next init) →
    reachable next init a.
```

Our soundness theorem expresses that all game states in an unfolded tree must be reachable in the game from the initial state.

```
Theorem unfold_tree_complete :
  ∀ {A : Type} (R : A → A → Prop) `{WellFounded R}
    (next : ∀ (a : A),
            {l : list A | Forall (λ x ⇒ R x a) l })
    (init : A),
  ∀ (a : A),
    reachable next init a →
    In_tree a (unfold_tree R next init).
```

Our completeness theorem expresses that any game state reachable from the initial state must be in the tree.

What to learn from the inductive proofs. The proofs for these theorems are not particularly complicated. While we will not display the full proofs here, they can be found in the supplementary material. In this section, we will present the distilled lessons we can learn from these proofs.

Soundness. The soundness proof is divided into two parts, just like the `unfold_tree` function. We state and prove soundness for the auxiliary function `unfold_tree_aux`, and invoke that proof from the original soundness proof. The soundness proof for the auxiliary function is by generalized induction on the accessibility predicate, which can be proved in Rocq more conveniently as a **`Fixpoint`**, since the `init` argument changes in recursive calls. Otherwise, this proof is relatively easy since it merely has to follow the recursive calls and keep track of reachability starting from the original `init` argument, the real initial state.

Completeness. The completeness proof is more complicated since there is no structure of recursive calls that we can directly unfold; we have to create our own structure. We can define a lemma that explicitly unfolds the tree only one step and handles all the associated proofs:

```
Lemma unfold_tree_unwrap :
  ∀ {A : Type} (R : A → A → Prop) `{WellFounded R}
    (next : ∀ (a : A),
            {l : list A | Forall (λ x ⇒ R x a) l })
    (init : A),
  unfold_tree R next init
= node init (map (unfold_tree R next) (next init).1).
```

We can now prove our completeness theorem by induction on the `reachable` parameter, where we can rewrite the goals using `unfold_tree_unwrap` and apply the inductive hypothesis when necessary.

> We defined `reachable` using the reflexive-transitive closure (`clos_refl_trans`), but the Rocq standard library also provides left-step (`clos_refl_trans_1n`) and right-step (`clos_refl_trans_n1`) variants. These variants are all equivalent but differ in convenience depending on the proof. For our completeness theorem, we use the left-step variant first and then switch back, to simplify the inductive proof.

5.2 Reasoning coinductively

Similar to `In` and `In_tree` for inductive lists and trees, we define `In_colist` and `In_cotree` for coinductive lists and trees. However, these relations are also defined *inductively*; they just happen to operate on coinductive values.

We also define a version of `step` and `reachable` that works on the type of `next` that we use in `unfold_cotree`:

```
Definition costep
           {A : Type}
           (next : A → colist A)
           (x y : A) : Prop :=
  In y (next x).
```

```
Definition coreachable
           {A : Type}
           (next : A → colist A)
         : A → A → Prop :=
  clos_refl_trans A (costep next).
```

Our soundness and correctness theorem statements for cotrees look very similar to the ones for trees:

```
Theorem unfold_cotree_sound :
  ∀ {A : Type} (next : A → colist A) (init : A),
  ∀ (a : A),
    In_cotree a (unfold_cotree next init) →
    coreachable next init a.
```

Our soundness theorem expresses that all game states in an unfolded tree must be reachable from the initial state. An interesting point here is that the In_cotree relation is inductive; it can only prove that a game state is in the tree using a finite derivation. The ∀ quantifier we have, however, allows us to express a property over all possible states in the game tree.[11]

```
Theorem unfold_cotree_complete :
  ∀ {A : Type} (next : A → colist A) (init : A),
  ∀ (a : A),
    coreachable next init a →
    In_cotree a (unfold_cotree next init).
```

Our completeness theorem expresses that any game state reachable from the initial state must be in the tree.

What to learn from the coinductive proofs. A 19th century poet once quipped that there are two things you do not want to see being made: laws and sausages. Coinductive proofs ought to be added to that list, as reasoning about coinductive types is much harder than reasoning about inductive types. We avoided the termination checker by going coinductive and defining our unfold_cotree function corecursively, but now we have to deal with the challenges of coinductive proofs, of which there are three major ones:

1. Corecursive function calls cannot easily be simplified in Rocq because reducing a possibly nonterminating term without any limits can cause an infinite loop.
2. Coinductive proofs do not compose, as it confuses Rocq's syntactic productivity checker.
3. We need bisimulation to reason about how possibly infinite values are related to each other; in particular, a bisimilarity relation for each coinductive type instead of Rocq's propositional equality type eq (or = in notation form).

[11] In fact, we can prove that this approach suffices to prove a property on all the states in a possibly infinite tree: We define a coinductive relation Forall_conodes, similar to the Forall relation on inductive lists in the Rocq standard library. We can prove that a property can be proven for anything that is In_cotree for that tree if and only if the Forall_conodes of that property can be proven. This proof is in the supplementary material.

While various extralinguistic solutions [28, 47, 13, 66, 30] have been devised to get around these challenges, here we will proceed by plain coinduction, as it is provided in Rocq [24], for easy readability of this proof pearl.

Dealing with simplification. Following the common advice in the Rocq community [5, 9], we define a special identity function cotree_decompose for our coinductive types that forces the coinductive value to decompose once:

```
Definition cotree_decompose
           {A : Type} (t : cotree A) :=
  match t with conode a f ⇒ conode a f end.
```

We then claim and prove an equality lemma cotree_decompose_eq that cotree_decompose is indeed the identity function specialized to cotrees:

```
Lemma cotree_decompose_eq :
  ∀ {A : Type} (t : cotree A), t = cotree_decompose t.
Proof. destruct t; auto. Qed.
```

This function and proof may look pointless, but it is actually quite helpful as we can use it to reduce coinductive values one step! We can simply **rewrite** with cotree_decompose_eq instead of resorting to a tactic like **simpl**, which does not work on coinductive values.

Defining bisimilarity. Proving the equality of possibly infinite values requires a possible infinite number of reductions. Unfortunately, Rocq's propositional equality type eq does not support that; eq depends on definitional equality of two terms, which requires normalization of the terms all at once. This may not be possible for coinductive values, therefore we define a new coinductive relation, called *bisimilarity* for each coinductive type whose "equality" we want to reason about. Here is the bisimilarity relation we define for colists:

```
CoInductive bisimilar_colist
            {A : Type} (R : A → A → Prop)
            : colist A → colist A → Prop :=
| conil_bisim  : bisimilar_colist R conil conil
| cocons_bisim :
    ∀ (a₁ a₂ : A) (l₁ l₂ : colist A),
    R a₁ a₂ →
    bisimilar_colist R l₁ l₂ →
    bisimilar_colist R (cocons a₁ l₁) (cocons a₂ l₂).
```

We have a constructor in our relation for each constructor of the coinductive type whose bisimilarity we are trying to express. Here we express that conil is bisimilar to another conil. We also express that cocons is bisimilar to cocons if their heads are related by R and their tails are bisimilar as well.

Why do we need R here? Well, our colist type is polymorphic; we do not know the type of data contained in a

colist. We want to accommodate both inductive and coinductive types in our reasoning. If we have values of an arbitrary inductive type foo in two colists, then we can use bisimilar_colist (@eq foo) to show bisimilarity of two such lists, where eq is Rocq's propositional equality type. If we have values of a coinductive type bar in two colists, then we will define a bisimilar_bar relation for the bar type, and use bisimilar_colist bisimilar_bar to show bisimilarity of such lists.

This flexibility in allowing a colist to contain elements of either inductive or coinductive types comes in handy when we try to define a bisimilarity relation for cotrees, because cotrees contain colists of cotrees. Here is how we define the bisimilarity relation for cotrees:

```
CoInductive bisimilar_cotree
            {A : Type} (R : A → A → Prop)
            : cotree A → cotree A → Prop :=
| conode_bisim :
    ∀ (a₁ a₂ : A) (f₁ f₂ : colist (cotree A)),
    R a₁ a₂ →
    bisimilar_colist (bisimilar_cotree R) f₁ f₂ →
    bisimilar_cotree R (conode a₁ f₁) (conode a₂ f₂).
```

A conode is bisimilar to another conode if the root labels are related by R and the children of the conode are bisimilar as well, according to bisimilar_colist and bisimilar_cotree.

Using bisimilarity. Now that we have bisimilarity relations for colists and cotrees, how do we use them in proofs? Rocq provides generalized rewriting mechanisms [12, 53] that allow us to rewrite with our bisimilarity relations, as if they were equalities. This convenience comes at the low cost of proving that our bisimilarity relations are equivalence relations, and that the predicates in which we want to rewrite respect the bisimilarity relations.

In_cotree predicate we defined earlier in subsection 5.2 is a good use case: We would like to prove that if two cotrees are bisimilar, then a label is in the first cotree if and only if it is in the second cotree. After proving this goal and registering it in Rocq using the **Add Parametric Morphism** command (or the Proper type class), we will be able to rewrite cotrees passed as an argument to In_cotree. Given that the return type of unfold_cotree_complete uses the In_cotree relation, we may (and do) want to rewrite the final goal using bisimilarity for cotrees.

In particular, having a bisimilarity proof of a single-step unfolding of cotrees is helpful:

```
Lemma unfold_cotree_unwrap :
  ∀ {A : Type} (next : ∀ (a : A), list A) (init : A),
    bisimilar_cotree (@eq A)
      (unfold_cotree next init)
      (conode init (comap (unfold_cotree next)
                          (next init))).
```

With the generalized rewriting of the final goal of unfold_cotree_complete with unfold_cotree_unwrap, the completeness proof is merely by induction on the left-step variant on the coreachable argument. Since the left-step variant (defined with clos_refl_trans_1n) is equivalent to the standard variant, our completeness proof is complete.

> From a proof-reuse perspective, the overlap between the inductive and coinductive developments is mostly at the level of *proof patterns*, rather than shared lemmas. The statements of soundness and completeness are deliberately parallel, but they live over different infrastructures: tree vs. cotree, In_tree vs. In_cotree, and equality vs. bisimilarity. As a result, proof terms cannot be shared directly, and we instead replicate the same shape of argument (one-step "unfold" lemma, induction on reachability, etc.) in each setting.

6 Building a game tree for tic-tac-toe

To demonstrate how useful our game tree library is, let us implement a simple, 3×3 tic-tac-toe game in Rocq. Here is how we define our game state:

```
Inductive player : Type := x | o.
Definition cell : Type := option player.
Inductive board : Type :=
| mkbd : cell → cell → cell →
         cell → cell → cell →
         cell → cell → cell → board.
Record game : Type :=
  { current_board : board ; next_turn : player }.
```

The initial state for the tic-tac-toe game can be defined as follows:

```
Definition ttt_init : game :=
  {| current_board := mkbd None None None
                           None None None
                           None None None
   ; next_turn := x |}.
```

Now, in order to build either the inductive or coinductive game tree for our tic-tac-toe game, we would need to define a function next that gives us the states one move away from any given game state. Suppose we already have a next : game → list game function. In order to use the unfold_tree function for inductive trees, we would need to define a relation and show that our next function gives us game states decreasing based on that relation. Ideally, we would have a relation that precisely captures the valid steps of the game, so that we can take advantage of the soundness and completeness proofs we worked on. We choose not to explain all the types, functions, and predicates involved, but here is the outline of the relation we defined for tic-tac-toe:

```
Inductive game_step : game → game → Prop :=
| gstep :
    ∀ (g : game) (m : move),
    get_result g = ongoing →
    valid_move g m →
    game_step g (apply_move g m).
```

The game_step relation states that if a game is not already won or at a draw, then the next player can make a move to an empty cell. Suppose we already have a ttt_next function that produces a list game from a game[12]. We will then need to prove that all members of the resulting list respect the game_step relation:

```
Lemma ttt_next_intrinsic :
  ∀ (g₁ : game),
  {l : list game | Forall (game_step g₁) l}.
Proof.
  intros g₁. exists (ttt_next g₁). (* ... *)
Defined.
```

Notice how easily we could fill the first half of the dependent pair with a call to ttt_next. The detour we took with zip_proofs and distribute in subsubsection 4.1.1 paid off here; as a user of our game tree library we did not have to deal with manipulating lists of dependent pairs, our library took care of that for us.

Do we now have all the pieces of the puzzle to construct a full, inductive game tree of our tic-tac-toe game? Almost! We can call unfold_tree, which requires a relation that governs how recursive calls are getting smaller. That relation is flip game_step, which switches the order of the arguments of game_step. The game_step relation takes the earlier game state as the first argument and the later game state as the second. This order is more natural for expressing state transitions, but less natural for expressing that something is getting "smaller," therefore we flip it for the termination relation:

```
Definition complete_tree : tree game :=
  unfold_tree (flip game_step)
              ttt_next_intrinsic
              ttt_init.
```

```
Error:
Term contains unresolved implicit arguments:
?WF: WellFounded (flip game_step)
(no type class instance found).
```

Oh, right! Our unfold_tree function requires the parametrized relation to have a WellFounded type class instance, but we forgot to prove the well-foundedness of our game_step relation! Given the complicated definition of game_step, how do we even approach such a proof? Proving well-foundedness of complicated relations can be unintuitive, but there is a trick up our sleeve that we may be able to use not only for tic-tac-toe, but for many other games as well: depicting our game_step relation as a *subrelation* of a simpler, more obviously well-founded relation. In tic-tac-toe, we can write a function that counts the number of empty cells on the board, and define a relation that states that the number of empty cells of one game board is less than that of the other game board:

```
Definition empty_cells (g : game) : nat := (* ... *).
Definition later (g₁ g₂ : game) : Prop :=
  empty_cells g₁ < empty_cells g₂.
```

We have an easier time proving the well-foundedness of the later relation, since the Rocq standard library already has a proof of the well-foundedness of the < relation for nats.

```
Instance WF_later : WellFounded later.
Proof.
  unfold later.
  apply wf_inverse_image, Nat.lt_wf_0.
Defined.
```

The last remaining step is to prove that game_step (or to be precise, its flipped version) is well-founded as well. But we know and can prove that a subrelation of a well-founded relation is also well-founded. If we can show that flip game_step is a subrelation of later, that is, if a game step from g_1 to g_2 implies that g_2 has fewer empty cells than g_1, we can use that proof to obtain the well-foundedness proof we need.

```
Instance WF_flip_game_step :
  WellFounded (flip game_step).
Proof.
  apply WF_subrelation, WF_later.
  intros g₂ g₁; inversion 1.
  apply less_empty_cells_after_apply_move; auto.
Defined.
```

Now we can define complete_tree with the same definition we had before, and use this game tree to implement an executable tic-tac-toe game for a human to play against our unbeatable AI. Our implementation uses a (yet unverified) generalized minimax algorithm over the finite game tree to always select an optimal move. This implementation can be found in our supplementary material, which can be run through Rocq's extraction to OCaml.

Building the tree coinductively. Alternatively, we could have defined complete_tree by first building the coinductive game tree, and then using the finiteness proof for our coinductively defined game to convert the cotree to a tree:

```
Definition complete_tree : tree game :=
  tree_of_cotree ttt_is_finite.1
                 (unfold_cotree ttt_next ttt_init).
```

[12] Also suppose we have a ttt_conext function that converts that the resulting list from ttt_next to a colist, for the sake of ttt_is_finite.

This approach to proving termination only requires us to know how many times the possibly infinite game tree has to be unwrapped. As we argued in subsection 3.1, this is easy to do for tic-tac-toe, but not for more complicated games.

7 SAT solving on a game tree

To demonstrate that our framework is not limited to traditional games, we also implemented a simple propositional SAT solver. The idea is to treat the problem of finding a satisfying assignment as a search game: the "moves" are truth assignments to propositional variables, and the outcome is whether the formula evaluates to true.

Formulas and assignments. We define an inductive type `formula` of propositional formulas with variables, Boolean literals, negation, conjunction, and disjunction. The free variables of a formula can be computed with the function `free_vars`, and evaluation function `eval` attempts to reduce a formula to a Boolean value if all variables have been assigned. A partial assignment is represented by a list of pairs `(name * bool)`, and applying an assignment to a formula replaces each variable by the corresponding Boolean, if present.

Moves and next states. From a game state consisting of the current partial assignment, the `sat_next` function chooses one of the remaining free variables and branches on the two possible Boolean values:

```
Definition sat_next
           (f : formula) (g : game) : list game :=
  match free_vars (apply_assignments f g) with
  | [] ⇒ []
  | n :: _ ⇒ [(n, true) :: g; (n, false) :: g]
  end.
```

As in tic-tac-toe, we can show that each such move strictly decreases the number of free variables (through a new `later` relation on assignments) so that recursion on this measure is well-founded. The intrinsic version `sat_next_intrinsic` packages recursive calls with required proofs of progress.

Folding over trees. In the tic-tac-toe example (§6) we built an interactive driver that queried the user at each node and rendered boards, so we wrote a custom, effectful traversal tailored to that user interface. In contrast, the SAT solver (§7) is non-interactive: there is no feedback loop with a user, and the goal is a single, aggregate result (a satisfying assignment or None). This makes a one-shot, pure traversal natural, so we introduce a generic fold over trees, `fold_tree`, and express the whole solver as a single fold over the unfolded game tree:

```
Fixpoint fold_tree
         {A B : Type}
         (g : A → list B → B)
         (t : tree A) : B :=
  match t with
  | node a f ⇒ g a (map (fold_tree g) f)
  end.
```

The idea is standard: given a node labelled by a value `a` and a list of subtrees `f`, we recursively fold each subtree and then combine the results with the user-supplied function `g`. Thus, `fold_tree` allows us to traverse the entire tree in a single pass and compute an aggregate result.

Searching the tree. We can now define the SAT solver in one step by folding over the tree generated from the initial formula `f`. Starting from the empty assignment, we unfold the tree with `unfold_tree`, and then fold it to look for a satisfying assignment:

```
Definition find_sat (f : formula) : option game :=
  fold_tree
    (λ g children ⇒
       match eval (apply_assignments f g) with
       | Some true ⇒ Some g
       | _ ⇒ List.hd_error (somes children)
       end)
    (unfold_tree (later f) (sat_next_intrinsic f) []).
```

At each node, if the current formula evaluates to `Some true` under the accumulated assignments, we return those assignments as a satisfying valuation. Otherwise, we examine the folded results of the children and propagate the first satisfying assignment found, if any. If no node yields `Some true`, the result is None, signifying that the formula is unsatisfiable.

Discussion. Conceptually, our solver is a baseline branch-on-variable search that resembles the branching skeleton of DPLL, but without unit propagation, pure-literal elimination, backjumping, or clause learning. We materialize the search space as an explicit tree and return the first assignment that makes the formula true. Thanks to our library, the structure mirrors tic-tac-toe: a next-state step, a well-founded measure for termination, and a fold to obtain a final result.

Operationally, this materialization happens only after code extraction. Rocq itself does not enforce an evaluation strategy, so both inductive and coinductive trees can, in principle, be consumed lazily. However, extraction to OCaml fixes an eager evaluation order, so the inductive version of our solver necessarily constructs the entire tree before folding it. If we had instead used `unfold_cotree`, or extracted to a lazy language such as Haskell, only the demanded prefix of the `tree` or `cotree` would be evaluated, making it possible to explore the search space incrementally rather than eagerly allocating it in full.

What we have verified here is that the search terminates, and that any returned assignment indeed satisfies the formula. Completeness for this baseline follows from exhaustively unfolding all free variables. By contrast, DPLL/CDCL implementations explore (and prune) the search space *implicitly*; verifying them requires separate proofs that each pruning step preserves satisfiability and that the overall driver terminates. In such settings, our framework can still be applied by unfolding a pruned next-state relation (one that incorporates the solver's simplification or propagation rules)

and then proving, in place of raw completeness, that each pruning step preserves soundness and does not remove any solution reachable by the unpruned game. Operationally, our approach can mimic the incremental, on-demand exploration used by practical pruned solvers (and avoid computing the full search space) either with inductive and coinductive trees in a lazy setting, or just coinductive trees in an eager setting.

8 Possible applications

Our solution is useful for any finite search space created by adding nontrivial transitions. These applications include games as well as pathfinding or constraint satisfaction problems. Here we suggest a few.

One area where game-tree finiteness is problematic is order matching in financial exchanges. To maximize traded quantity, we may represent possible order matches as a game tree. Matching buy and sell orders of unequal size leaves a residual order that returns to the book, complicating proofs of decreasing size. We can address this by defining a relation on order-book potential, showing it decreases with matches and is well-founded. This approach can help factor out the proofs about recursive call tree traversals from proofs about matching if applied to existing work [50, 19, 20, 18].

A similar problem arises if we want to use a game tree to represent all β-reductions of a simply-typed lambda calculus [10] term. Since some β-reductions require calling the substitution function, convincing Rocq that the term is getting smaller is not trivial. If we wanted to build a tree of all possible reductions of a term, however, we can use a `flip`ped version of the β-reduction relation $t_1 \to^* t_2$ as an argument to our `unfold_tree` function. Showing the well-foundedness of that relation amounts to a constructive proof that β-reduction in simply-typed lambda calculus is strongly normalizing [58, 17].

Game trees can also model constraint-search procedures. In SAT, classical DPLL [15, 14] and modern CDCL [51] are often pictured as trees whose nodes branch on a Boolean variable, even though practical solvers do not construct the whole tree. Our SAT section (§7) instantiates this picture *explicitly* as a verified baseline: a full unfold with no pruning. To approach verified DPLL/CDCL, one can instead unfold a *pruned* game tree whose edges implement unit propagation, pure-literal elimination, backjumping, and clause learning, and then fold with correctness invariants. Formally verified implementations face termination obligations: either proving a decreasing measure (e.g., number of unassigned variables) [26], or using the fuel pattern for nonstructural recursion [29]. In a different tradition of logic, computation trees are used to reason about time [11]. Since computation trees can be potentially infinite, O'Connor [44] explores an implementation of them as coinductive rose trees (and a coinductive unfold on them to build a model), and describes a proven-sound, bounded proof search procedure for computation tree logic using them.

9 Related work

Nipkow et al. [40] and Nipkow [39] verify alpha-beta pruning for game trees in Isabelle/HOL [41]. Their concern is algorithmic search over a tree given as input, formalized in a classical higher-order logic with built-in support for partial functions and (co)datatypes. In contrast, our paper focuses on the construction of the game tree itself in Rocq, a dependent type theory where all functions must be total and termination or productivity must be justified explicitly (via well-founded recursion or guarded corecursion). Thus the challenges are quite different: Isabelle developments lean on partiality and automation for reasoning about search procedures, whereas our development emphasizes intrinsic total definitions and explicit termination/productivity proofs. In this sense, the two approaches are complementary, tackling different aspects of reasoning about game trees.

Escardó and Oliva [16] explore a formalization of game trees different than ours and Nipkow [39]'s. They model games in the Agda proof assistant [43] as well-founded dependent trees (W-types) with history-dependent branching and use selection functions to compute optimal strategies by backward induction. In contrast, we work in Rocq with polymorphic (but not dependent) rose trees built from a `next` function, focusing on constructing the tree itself correctly (both inductively and coinductively).

Alexandru et al. [2] use bialgebraic semantics to define intrinsically correct sorting algorithms, by combining folds and unfolds of lists. They implement *treesort* [62], which reifies the call tree as a binary search tree. They use a termination measure as well, but for a custom binary tree type that is descending by construction. They do not cover rose trees or reason about games, but we look forward to a bialgebraic take on game trees.

Coinductive game trees handle games regardless of termination: infinite moves, infinite depth, and non-monotonic `next` functions are all allowed. Inductive trees, however, require the `next` function to produce strictly "lesser" states at each step. For some games, this may be too restrictive, motivating the use of well-quasi-orders instead. Intuitively, these allow occasional increases in state size, as long as states eventually become smaller. Vytiniotis et al. [65] provide a blueprint for proving termination under conditions similar to well-quasi-orders.

Hydra battles, in which the game state is itself a finite rose tree, are a common example where termination is highly nontrivial: cutting off a head triggers regrowth, and proving that all such battles end requires arguments that go far beyond simple structural size measures. Castéran et al. [8] formalize this proof in Rocq. While this makes hydras an interesting instance of rose tree games, our focus here is different:

We study how to construct and reason about game trees in general (via inductive and coinductive unfolds), rather than proving termination of a particular regrowth dynamic.

10 Conclusion

We compared two approaches to verifiably building game trees in Rocq: an inductive method (for finite trees) and a coinductive method (for possibly infinite trees). While inductive rose trees require careful proofs of termination via well-foundedness, they allow straightforward reasoning about properties of the resulting trees. Coinductive rose trees, on the other hand, are simpler to define but require more complex coinductive proofs to reason about infinite structures.

In both cases, we showed how to prove soundness (every node in the tree represents a reachable state) and completeness (every reachable state appears in the tree). We demonstrated the real-world applicability of our rose tree functions by implementing a verified tic-tac-toe game tree and a SAT solver, but this approach generalizes to other game and search problems.

Acknowledgments

AI tools were used to reword sentences idiomatically, to check the grammar and spelling, and to complete a few of the Rocq proofs in the supplementary material. We also thank Joseph W. Cutler, Bhargav Shivkumar, Julien Vanegue, Matthew Z. Weaver, and the anonymous reviewers for their comments.

References

[1] Victor Adamchik. 2009. Game trees. http://web.archive.org/web/20240419134929/https://viterbi-web.usc.edu/~adamchik/15-121/lectures/Game%20Trees/Game%20Trees.html

[2] Cass Alexandru, Vikraman Choudhury, Jurriaan Rot, and Niels van der Weide. 2025. Intrinsically Correct Sorting in Cubical Agda. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Denver, CO, USA) *(CPP '25)*. Association for Computing Machinery, New York, NY, USA, 34–49. doi:10.1145/3703595.3705873

[3] Antonia Balaa and Yves Bertot. 2000. Fix-Point Equations for Well-Founded Recursion in Type Theory. In *Theorem Proving in Higher Order Logics*, Mark Aagaard and John Harrison (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[4] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. 2006. Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant. In *Functional and Logic Programming*, Masami Hagiya and Philip Wadler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 114–129.

[5] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag.

[6] Richard Bird. 2010. *Pearls of functional algorithm design*. Cambridge University Press.

[7] Ana Bove, Alexander Krauss, and Matthieu Sozeau. 2016. Partiality and recursion in interactive theorem provers–an overview. *Mathematical Structures in Computer Science* 26, 1 (2016), 38–88. doi:10.1017/S0960129514000115

[8] Pierre Castéran, Jérémy Damour, Karl Palmskog, Clément Pit-Claudel, and Théo Zimmermann. 2022. Hydras & Co.: Formalized mathematics in Coq for inspiration and entertainment. In *Journées Francophones des Langages Applicatifs: JFLA 2022*. St-Médard d'Excideuil, France. https://hal.science/hal-03404668

[9] Adam Chlipala. 2011. *Certified Programming with Dependent Types*. MIT Press.

[10] Alonzo Church. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5, 2 (1940), 56–68. doi:10.2307/2266170

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (apr 1986), 244–263. doi:10.1145/5397.5399

[12] Claudio Sacerdoti Coen. 2004. A semi-reflexive tactic for (sub-)equational reasoning. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs* (Jouy-en-Josas, France) *(TYPES'04)*. Springer-Verlag, Berlin, Heidelberg, 98–114. doi:10.1007/11617990_7

[13] Łukasz Czajka. 2019. First-Order Guarded Coinduction in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:18. doi:10.4230/LIPIcs.ITP.2019.14

[14] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (jul 1962), 394–397. doi:10.1145/368273.368557

[15] Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (jul 1960), 201–215. doi:10.1145/321033.321034

[16] Martín Escardó and Paulo Oliva. 2023. Higher-order games with dependent types. *Theoretical Computer Science* 974 (2023), 114111. doi:10.1016/j.tcs.2023.114111

[17] Proof Assistants Stack Exchange. 2022. Constructive proof of strong normalization for simply typed lambda calculus. https://web.archive.org/web/20240717155732/https://proofassistants.stackexchange.com/questions/1135/constructive-proof-of-strong-normalization-for-simply-typed-lambda-calculus

[18] Mohit Garg, N. Raja, Suneel Sarswat, and Abhishek Kr Singh. 2024. Double Auctions: Formalization and Automated Checkers. *CoRR* abs/2410.18751 (2024). arXiv:2410.18751 doi:10.48550/ARXIV.2410.18751

[19] Mohit Garg and Suneel Sarswat. 2022. The Design and Regulation of Exchanges: A Formal Approach. In *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 250)*, Anuj Dawar and Venkatesan Guruswami (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 39:1–39:21. doi:10.4230/LIPIcs.FSTTCS.2022.39

[20] Mohit Garg and Suneel Sarswat. 2024. Efficient and Verified Continuous Double Auctions. arXiv:2412.08624 [cs.LO] doi:10.48550/arXiv.2412.08624

[21] Jeremy Gibbons. 1991. *Algebras for tree algorithms*. Ph. D. Dissertation. University of Oxford. https://web.archive.org/web/20240605140628/https://www.cs.ox.ac.uk/files/3422/PRG94.pdf

[22] Jeremy Gibbons. 2000. Generic downwards accumulations. *Science of Computer Programming* 37, 1 (2000), 37–65. doi:10.1016/S0167-6423(99)00022-2

[23] Jeremy Gibbons and Geraint Jones. 1998. The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 273–279. doi:10.1145/289423.289455

[24] Eduardo Giménez. 1996. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In *Types for Proofs and Programs*, Stefano Berardi and Mario Coppo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 135–152.

[25] Jennifer Hackett, Graham Hutton, and Mauro Jaskelioff. 2013. The Under-Performing Unfold: A new approach to optimising corecursive programs. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages* (Nijmegen, Netherlands) *(IFL '13)*. Association for Computing Machinery, New York, NY, USA, 4321–4332. doi:10.1145/2620678.2620679

[26] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 117 (oct 2021), 30 pages. doi:10.1145/3485494

[27] Feng-hsiung Hsu, Murray S. Campbell, and A. Joseph Hoane. 1995. Deep Blue system overview. In *Proceedings of the 9th International Conference on Supercomputing* (Barcelona, Spain) *(ICS '95)*. Association for Computing Machinery, New York, NY, USA, 240–244. doi:10.1145/224538.224567

[28] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The power of parameterization in coinductive proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 193–206. doi:10.1145/2429069.2429093

[29] Ende Jin. 2022. Parametrized CDCL Verified in Coq. doi:10.13140/RG.2.2.34926.54085

[30] Jaewoo Kim, Yeonwoo Nam, and Chung-Kil Hur. 2025. Coco: Corecursion with Compositional Heterogeneous Productivity. arXiv:2511.21093 [cs.LO] https://arxiv.org/abs/2511.21093

[31] Joomy Korkut, Maksim Trifunovski, and Daniel Licata. 2016. Intrinsic verification of a regular expression matcher. (2016). http://web.archive.org/web/20230605134641/http://dlicata.wescreates.wesleyan.edu/pubs/ktl16regexp/ktl16regexp.pdf

[32] François Labelle. 2015. The longest possible chess game, and bounds on the number of possible chess games. http://web.archive.org/web/20240326160632/https://wismuth.com/chess/longest-game.html

[33] Xavier Leroy. 2024. Well-founded recursion done right. In *CoqPL 2024: The Tenth International Workshop on Coq for Programming Languages*. https://web.archive.org/web/20240718174458/https://xavierleroy.org/publi/wf-recursion.pdf

[34] Conor McBride. 2015. Turing-completeness totally free. In *International Conference on Mathematics of Program Construction*. Springer, 257–275. doi:10.1007/978-3-319-19797-5_13

[35] Lambert Meertens. 1988. First steps towards the theory of rose trees. *CWI, Amsterdam* (1988). https://www.academia.edu/29461542/First_steps_towards_the_theory_of_rose_trees

[36] Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on functional programming languages and computer architecture*. Springer, 124–144. doi:10.1007/3540543961_7

[37] Kosuke Murata and Kento Emoto. 2019. Recursion Schemes in Coq. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 202–221. doi:10.1007/978-3-030-34175-6_11

[38] Tom Murphy, VII. 2020. Is this the longest Chess game?. In *SIGBOVIK*. http://web.archive.org/web/20240206053907/http://tom7.org/chess/longest.pdf

[39] Tobias Nipkow. 2024. Alpha-Beta Pruning Verified. In *15th International Conference on Interactive Theorem Proving (ITP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 309)*, Yves Bertot, Temur Kutsia, and Michael Norrish (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:4. doi:10.4230/LIPIcs.ITP.2024.1

[40] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gomez-Londono, Peter Lammich, Christian Sternagel, Simon Wimmer, and Bohua Zhan. 2024. *Functional Data Structures and Algorithms. A Proof Assistant Approach*. https://fdsa-book.net

[41] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg.

[42] Bengt Nordström. 1988. Terminating general recursion. *BIT Numerical Mathematics* 28 (1988), 605–619. doi:10.1007/BF01941137

[43] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Chalmers University of Technology.

[44] Liam O'Connor. 2016. Applications of applicative proof search. In *Proceedings of the 1st International Workshop on Type-Driven Development* (Nara, Japan) *(TyDe 2016)*. Association for Computing Machinery, New York, NY, USA, 43–55. doi:10.1145/2976022.2976030

[45] Zoe Paraskevopoulou and Cătălin Hriţcu. 2014. A Coq framework for verified property-based testing. https://web.archive.org/web/20241014210851/https://catalin-hritcu.github.io/publications/verified-testing-report.pdf

[46] Chris Piech. 2012. Deep Blue. http://web.archive.org/web/20231127170321/https://stanford.edu/~cpiech/cs221/apps/deepBlue.html

[47] Damien Pous. 2016. Coinduction All the Way Up. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 307–316. doi:10.1145/2933575.2934564

[48] Simon Robillard. 2014. Catamorphism generation and fusion using Coq. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 180–185. doi:10.1109/SYNASC.2014.32

[49] The Rocq Team. 2025. The Rocq Prover. http://rocq-prover.org

[50] Suneel Sarswat and Abhishek Kr Singh. 2020. Formally Verified Trades in Financial Markets. In *Formal Methods and Software Engineering*, Shang-Wei Lin, Zhe Hou, and Brendan Mahony (Eds.). Springer International Publishing, Cham, 217–232. doi:10.1007/978-3-030-63406-3_13

[51] João P. Marques Silva and Karem A. Sakallah. 1997. GRASP—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design* (San Jose, California, USA) *(ICCAD '96)*. IEEE Computer Society, USA, 220–227. doi:10.1109/ICCAD.1996.569607

[52] Matthieu Sozeau. 2006. Subset coercions in Coq. In *International Workshop on Types for Proofs and Programs*. Springer, 237–252.

[53] Matthieu Sozeau. 2009. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning* 2, 1 (Jan. 2009), 41–62. doi:10.6092/issn.1972-5787/1574

[54] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1, Article 8 (Jan. 2025), 74 pages. doi:10.1145/3706056

[55] Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP, Article 86 (jul 2019), 29 pages. doi:10.1145/3341690

[56] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293. doi:10.1007/978-3-540-71067-7_23

[57] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. doi:10.1145/3167092

[58] W. W. Tait. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32, 2 (1967), 198–212. doi:10.2307/2271658

[59] Yee-Jian Tan. 2024. Towards Formalising the Guard Checker of Coq. https://web.archive.org/web/20241014164523/https://www.yeejian.dev/files/240827-report.pdf

[60] Enrico Tassi. 2019. Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. In *ITP 2019-10th International Conference on Interactive Theorem Proving*. doi:10.4230/LIPIcs.ITP.2019.29

[61] Alan Mathison Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math* 58, 345-363 (1936), 5. doi:10.1112/plms/s2-42.1.230

[62] D. A. Turner. 1995. Elementary strong functional programming. In *Functional Programming Languages in Education*, Pieter H. Hartel and Rinus Plasmeijer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–13. doi:10.1007/3-540-60675-0_35

[63] Marcel Ullrich. 2020. Generating induction principles for nested inductive types in MetaCoq. https://web.archive.org/web/20240812182346/https://www.ps.uni-saarland.de/~ullrich/bachelor/thesis.pdf

[64] Tarmo Uustalu and Varmo Vene. 1999. Primitive (co) recursion and course-of-value (co) iteration, categorically. *Informatica* 10, 1 (1999), 5–26. doi:10.3233/INF-1999-10102

[65] Dimitrios Vytiniotis, Thierry Coquand, and David Wahlstedt. 2012. Stop when you are almost-full: Adventures in constructive termination. In *Interactive Theorem Proving: Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings 3*. Springer, 250–265. doi:10.1007/978-3-642-32347-8_17

[66] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 71–84. doi:10.1145/3372885.3373813

Received 2025-09-11; accepted 2025-11-13