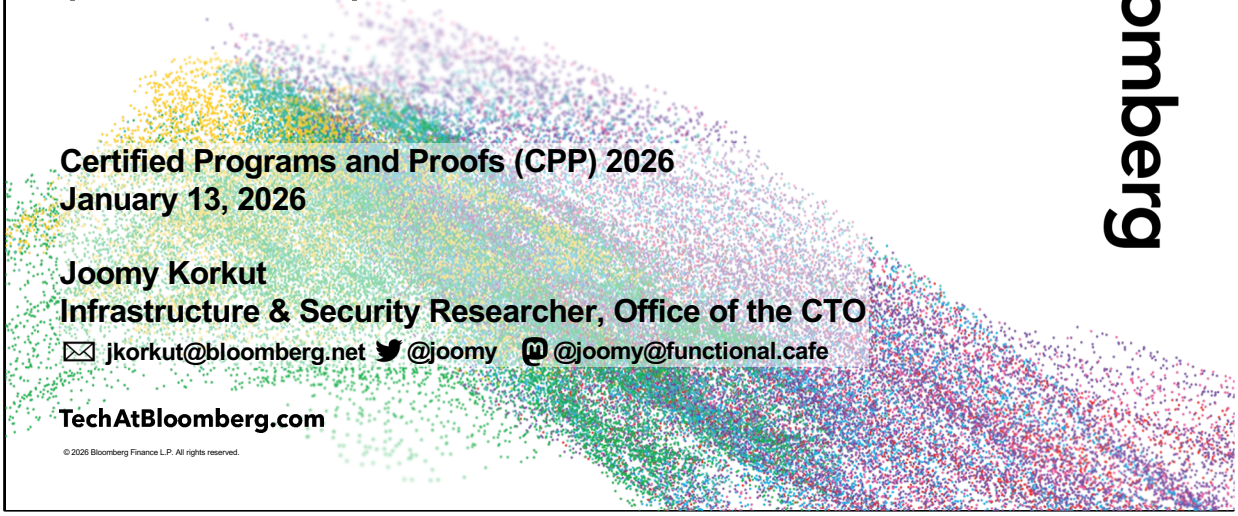
A Rose Tree is Blooming
(Proof Pearl)

Bloomberg



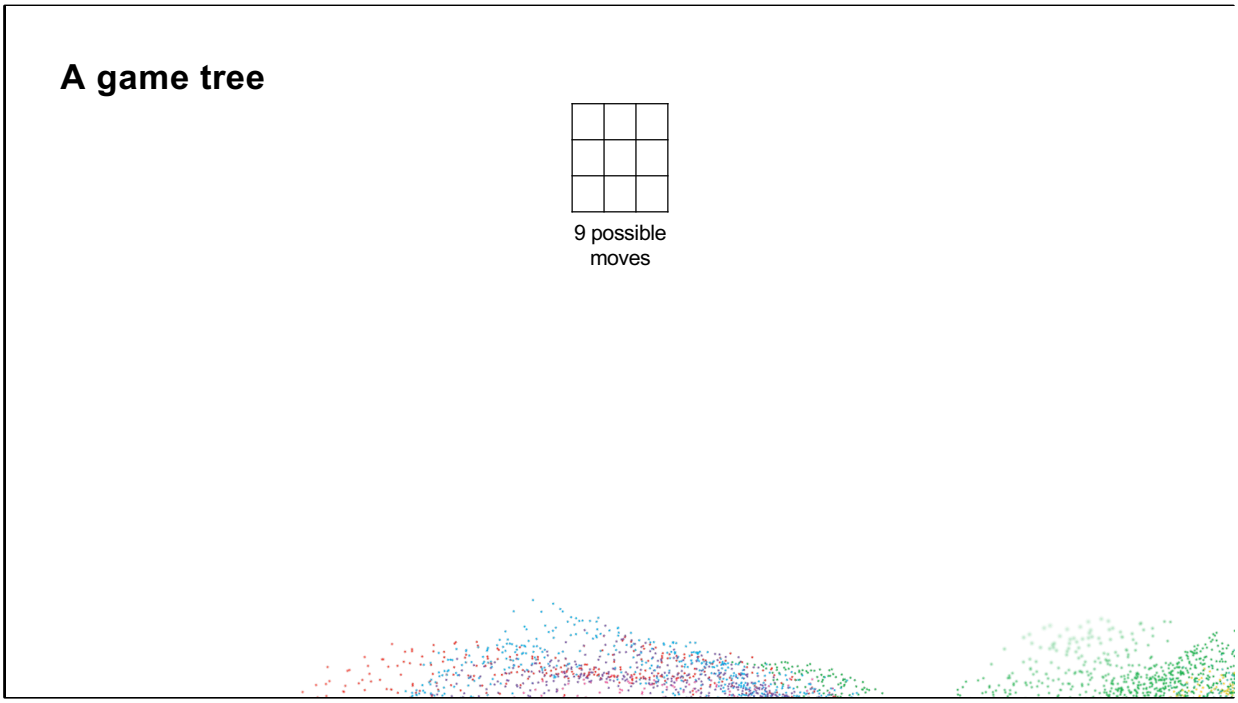
Certified Programs and Proofs (CPP) 2026
January 13, 2026

Joomy Korkut
Infrastructure & Security Researcher, Office of the CTO
jkorkut@bloomberg.net @joomy @joomy@functional.cafe

TechAtBloomberg.com

Hi everyone!
I'm Joomy, a researcher at Bloomberg's CTO Infrastructure and Security Research team in New York City.

Today I am going to present you a proof pearl about
- game trees,
- implementing game trees as rose trees in functional languages, particularly the Rocq proof assistant
- and more importantly, recursively building game trees, correctly.

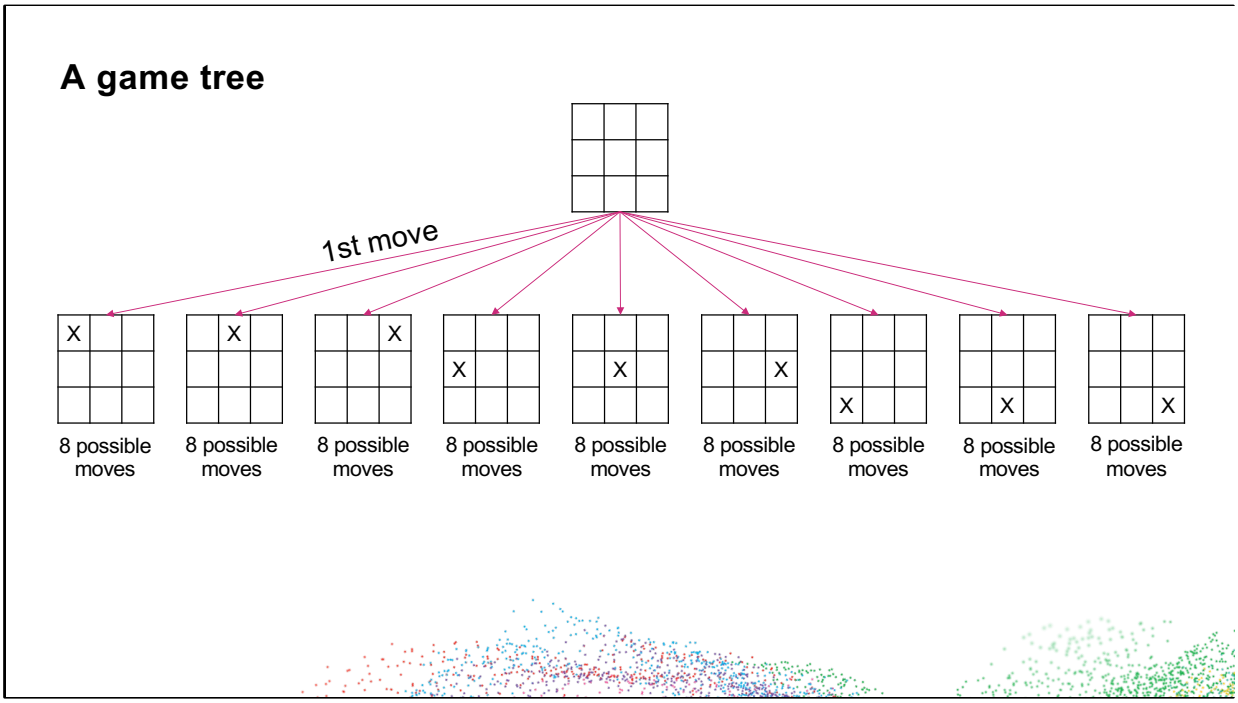


Let’s start with the very basics. What are game trees, what are they good for?

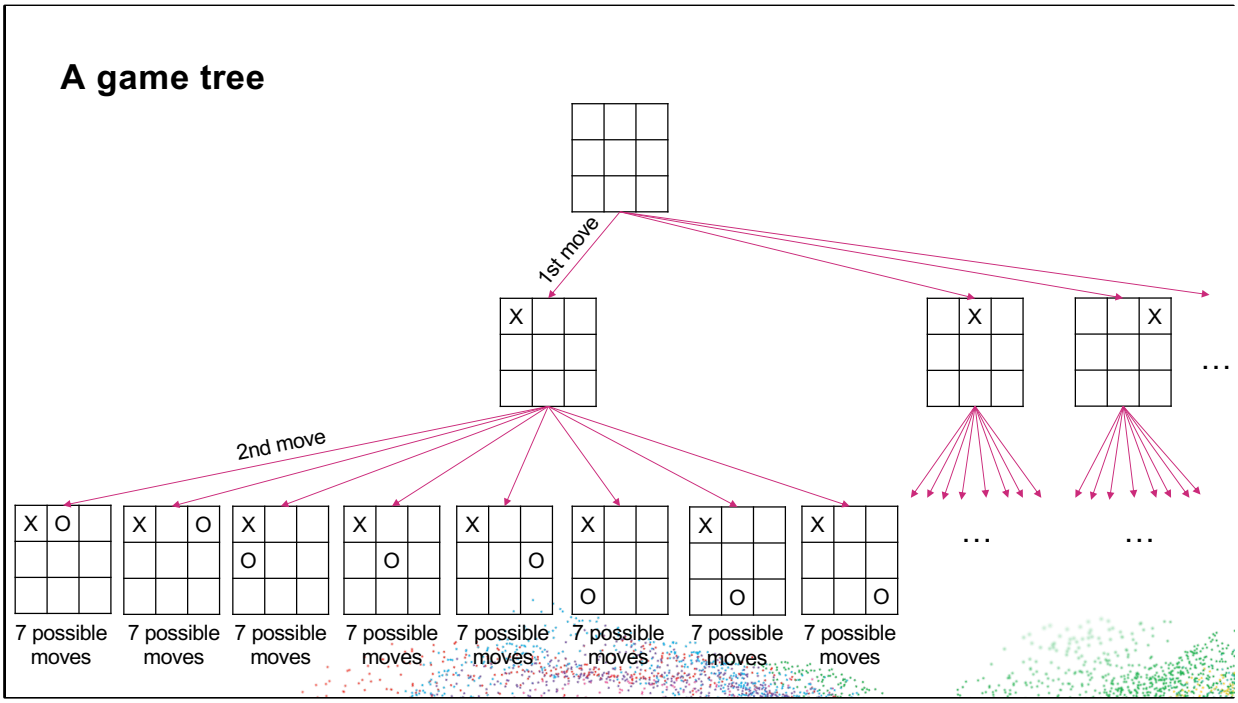
Game trees are a mathematical abstraction in game theory. They are used to reason about games and searching for the best move in a game. Famously DeepBlue, a program that did game tree search for chess, won against Kasparov in the 90s. Game trees can also be used for pathfinding and constraint satisfaction.

Game trees depict the state space of a game. A game tree consists of states of a game organized as a tree, where each branch leads to a state after a move.
Let’s see an example.

Consider tic-tac-toe. We want to explore the state space of tic-tac-toe.
The tree has the empty board at the root. Now, player 1 has to mark X in one of the 9 empty cells. That means, there are 9 possible moves they can make, and that there should be 9 branches of the root.



After the first move, this is what the game tree looks like.
Now, at each of these new boards, we know it's the second player's turn.
In their turn, player 2 will have to mark O in one of the empty cells. Given that player 2 has 8 possible moves at each of these new boards, these new boards must each have 8 branches.

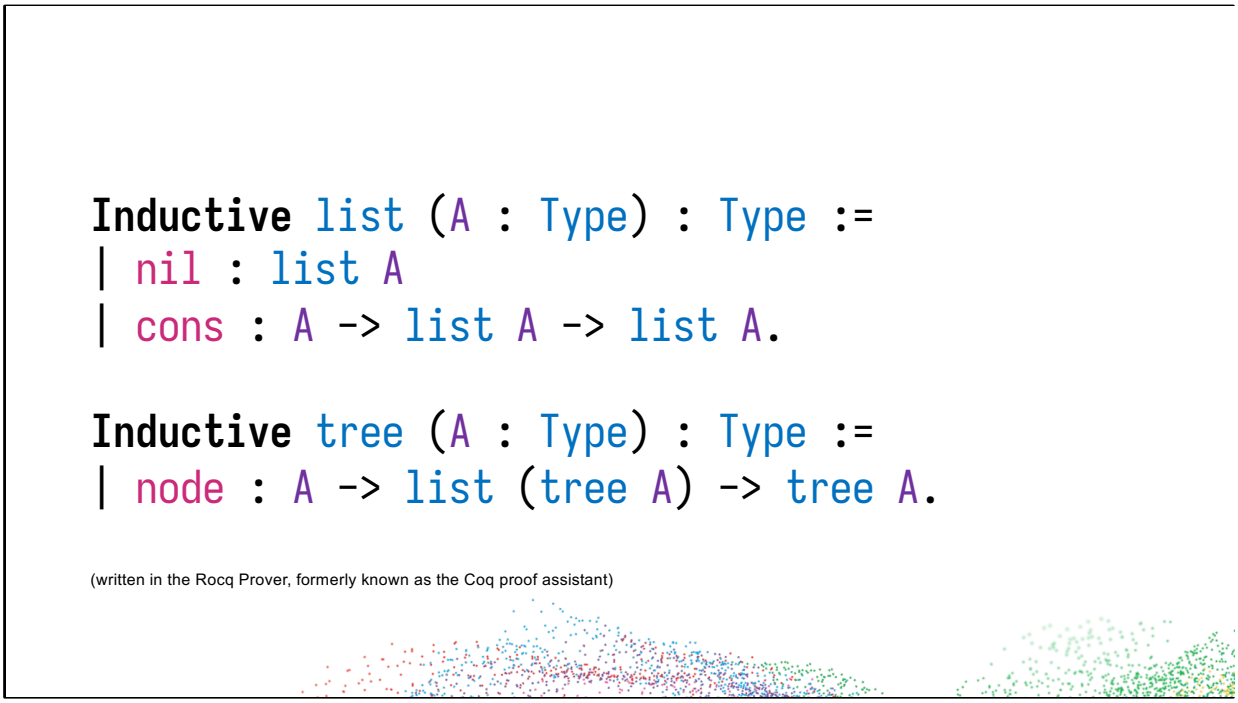


I don't have enough room here to show the entire tree, so here is only the leftmost branch expanded. At each of these new boards, it is player 1's turn again, and the game keeps going.

I think you get the idea. The entire tree, assuming that we stop expanding a branch after a game is won, is around 550 thousand nodes, so that wouldn't be the best use of our time here. ☺

Now, let's get back to programming!

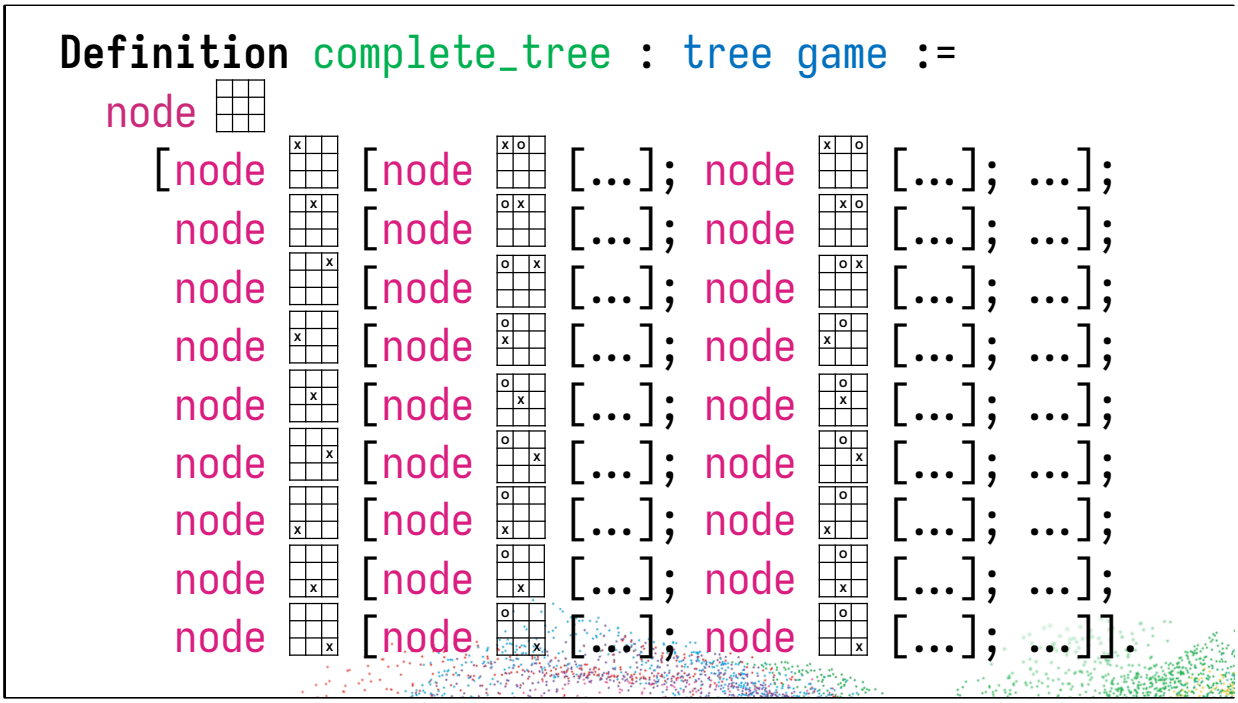
There are various data structures we can use to represent game trees, but the most idiomatic and common one for functional programming is rose trees.



For those who are unfamiliar, a rose tree is a tree data structure, in which a node contains a list of subtrees (as opposed to only 2 subtrees like in a binary tree)

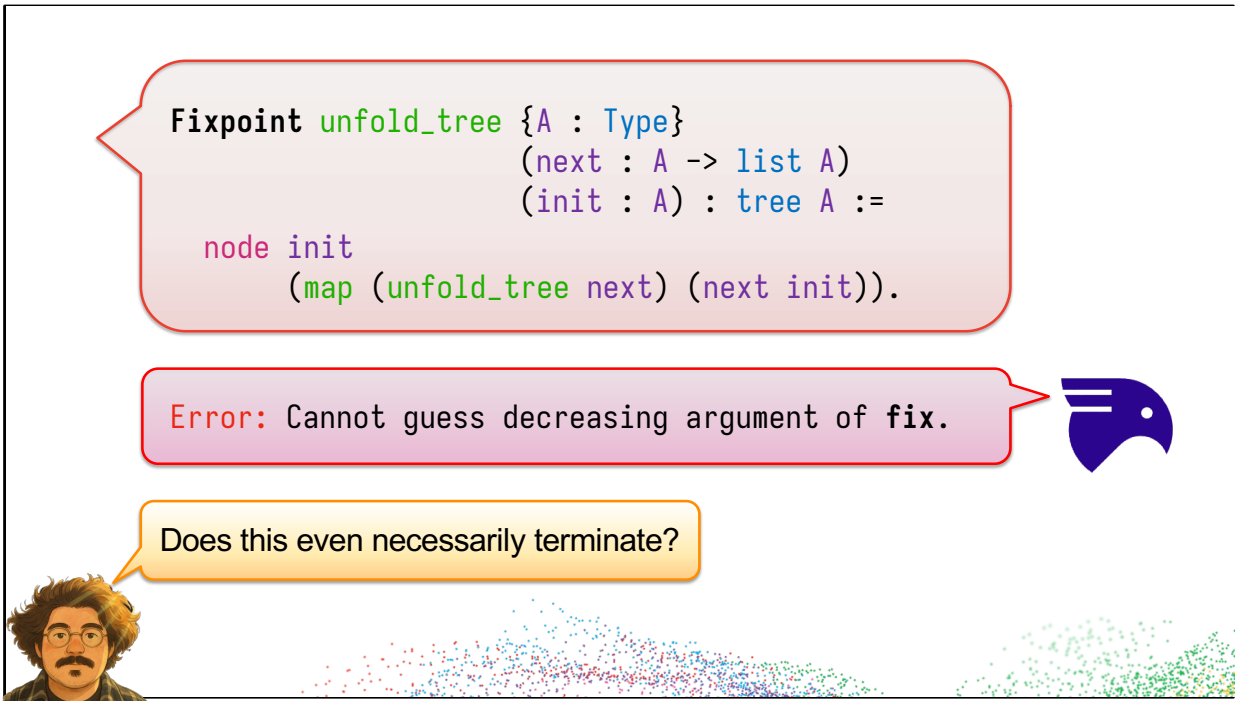
Here in the slide, we see the definition of rose trees in Rocq.
First, we have the definition of lists (linked lists), a list can either be nil, an empty list, or cons, a list that contains one elements and another sublist.

After lists, we define the type of trees. A tree can only be a node that has a root element, and a list of subtrees.



This is the game tree for tic-tac-toe that we saw before, but now written as a rose tree.

Obviously we can write it out by hand like this, but can we write a function that builds this tree for us?



We want to make our function parametric over different types of game states, that is the type A here.
Then we have the `next` parameter: a function that takes a state, and gives us the list of states one move away from that.
And then we have the `init` parameter: the initial state of the game. I should be able to compute the entire state space from this, right?

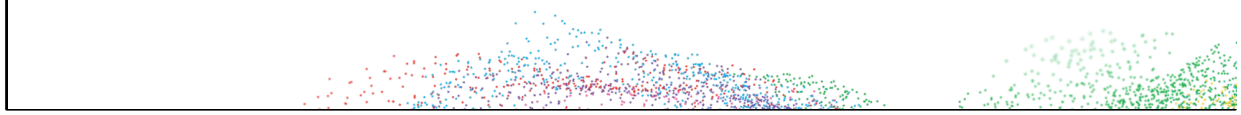
<click> Let's attempt writing this function. The first solution here is to put the initial state in a node, get the states after one move from the initial node, and then map over the next states with recursive calls. Great, we must be done! Just to make sure, let's check what Rocq says...
<click> Oof! Rocq is not okay with this function definition. It says it cannot guess the decreasing argument of fix. What does that mean? Well, it means Rocq is not convinced that this function terminates. Rocq's termination checker often stops you when a recursive function is not obviously terminating. What do we have to do to convince Rocq?

<click> Though I have to interrupt this line of thinking and ask: Does this even necessarily terminate? For tic-tac-toe, sure, it will, but what about other

games? Not all games are finite like tic-tac-toe. If we are trying to explore the state space of an infinite game, then the game tree for that will NOT be finite!

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.

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



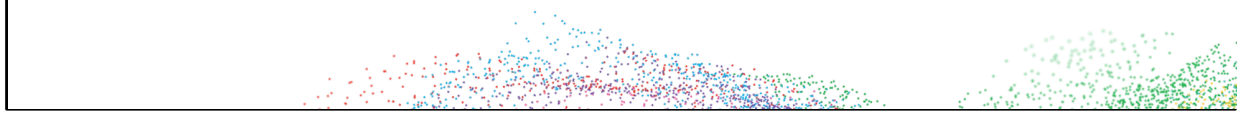
That's a bit of a problem for us, because we defined rose trees in Rocq with in**duc**tive types. Inductive values are finite, and recursive functions in Rocq must terminate! Where does that leave us? Now we have two problems we need to deal with:

1. How do we write the recursive function we want to write?
2. How do we represent and compute infinite trees? Especially if we want to deal with infinite games.

There is a way we can answer both of these questions..

```
CoInductive colist (A : Type) : Type :=
| conil : colist A
| cocons : A -> colist A -> colist A.

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

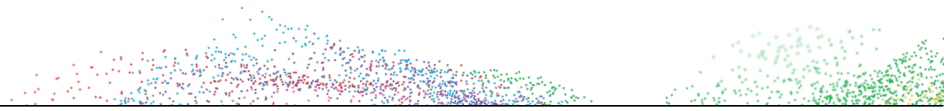


And that is coinductive types!

With inductive types and recursive functions, we could only have finite values and terminating functions.
With coinductive types and corecursive functions, we have possibly infinite values, and possibly nonterminating functions.

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

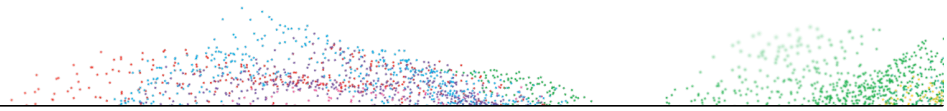
```
Error: Cannot guess decreasing argument of fix.
```



Let's look at our failed attempt of writing unfold for **inductive** trees… and make it work on **coinductive** trees….

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

```
unfold_cotree is corecursively defined.
```



And that works just fine!

Those of you who have dealt with coinductive values and tried to reason about them might say, but Joomy, we could have had a finite tree, and it would be a lot easier to write proofs about them. Do not worry...

```
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 []
    | S fuel' =>
        node a (map (tree_of_cotree fuel')
                    (list_of_colist fuel f))
    end
  end.
```

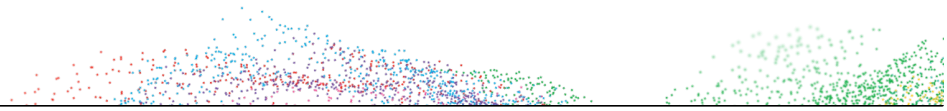
Even if we have a possibly infinite cotree, we can still convert that to a finite tree by taking a finite number of elements from the cotree.

<click> This is similar to what proof assistant folks call the fuel pattern! We have a program that may or may not terminate, so we add a fuel parameter and let the program run as long as there is some fuel.

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


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


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



Now, using the function to convert a cotree to a tree, we can say, a tree is finite if at some point, adding more fuel doesn't change the tree that you get.

<click> Using that definition, we can even say a game, as defined by its initial state and next states function, is finite if the unfolded game tree is finite.

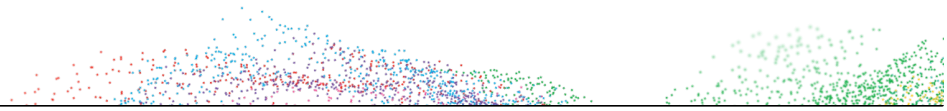
<click> And using that, we can prove that tic-tac-toe is a finite game! We need to unroll the game tree 10 times, one for the empty board, and 9 because the longest game of tic-tac-toe takes 9 moves.

Lessons to learn:

- Coinductive types can be infinite, so no need for termination checking.
- With no termination checker, writing unfold gets **easier**.

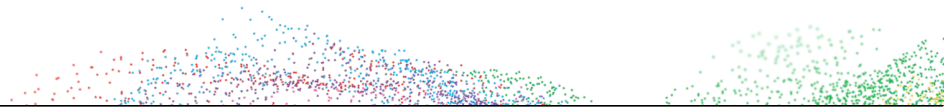
Lessons we do not have time for today, but are in the paper:

- Reasoning about coinductive values is much **harder and verbose**!
  (~450 LoC for inductive vs ~900 LoC for coinductive)
- Reuse of coinductive proofs in Rocq is **annoying** (but one can get around it).
- We need to use techniques like **bisimulation**.



```
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.
```



Let's get back to inductive game trees! I believe we can still rescue the function we had before.

We had a function that we
1) cannot convince Rocq that it is terminating
2) actually does not terminate for infinite games.

Okay, so let's write a version that works for finite games!

```
Fixpoint unfold_tree
         {A : Type}
         (R : A -> A -> Prop)
         (next : ∀ (a₁ : A),
                 { l : list A | Forall (λ a₂ => R a₂ a₁) l })
         (init : A) : tree A :=
  node init
       (map (unfold_tree R next) (next init).1).
```

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

Here is an attempt! This is very similar to the version we had before, except now we have...
<click> a relation R,
<click> and our `next` function returns new game states that are “less” than the previous states according to that relation.

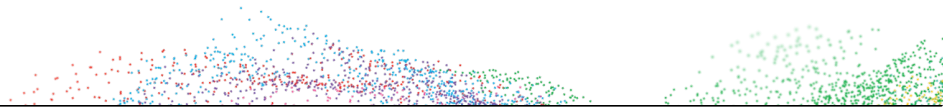
Except...
<click> That is not enough to convince Rocq. We need to
1) prove that this relation is well-founded,
2) use the well-foundedness of this relation to convince Rocq that this function terminates.

Well-foundedness of a relation

$$0 < 1 < 2 < 4 < 8 < 16 < 32$$

$$\emptyset \subset \{a\} \subset \{a, b\}$$

$$? \ldots a_6 \ R \ a_5 \ R \ a_4 \ R \ a_3 \ R \ a_2 \ R \ a_2 \ R \ a_1$$

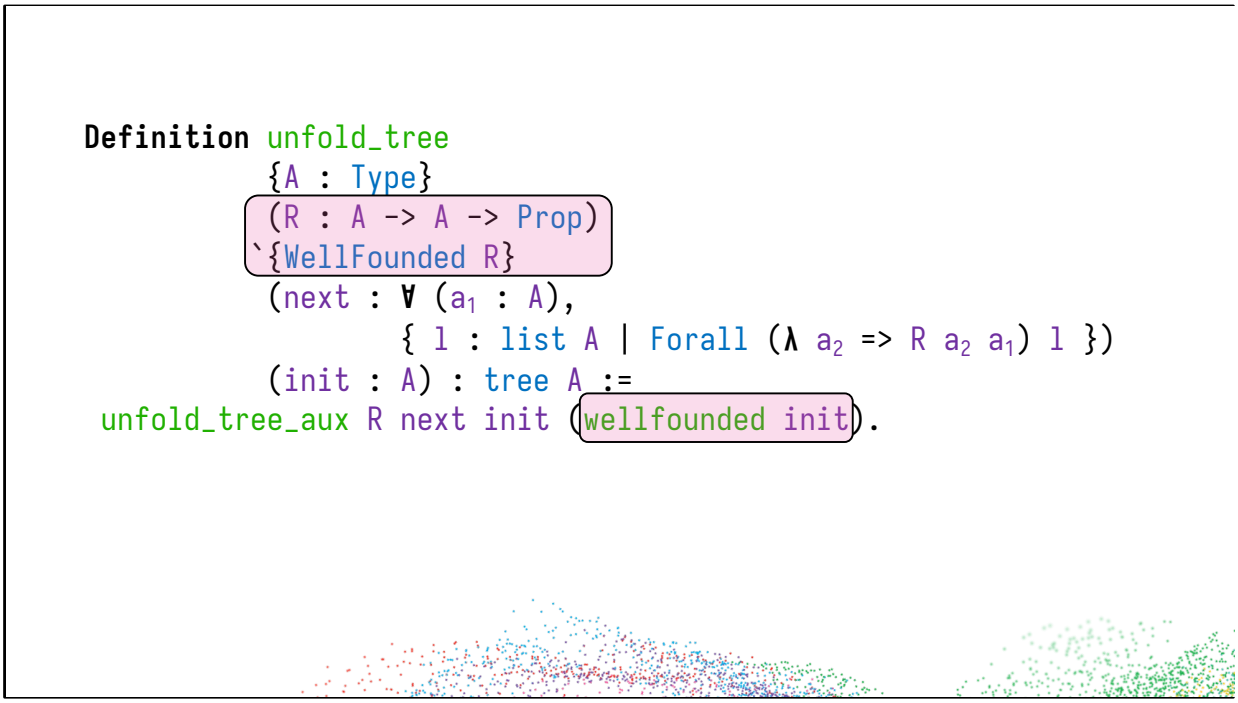


What do I mean by well-foundedness? I mean that **things should not be able to keep getting smaller forever**. In other words, it is impossible to have an infinite decreasing chain.

Take the less than relation on natural numbers. You cannot build an infinite decreasing chain because you cannot get smaller than zero. So less than on natural numbers is well-founded

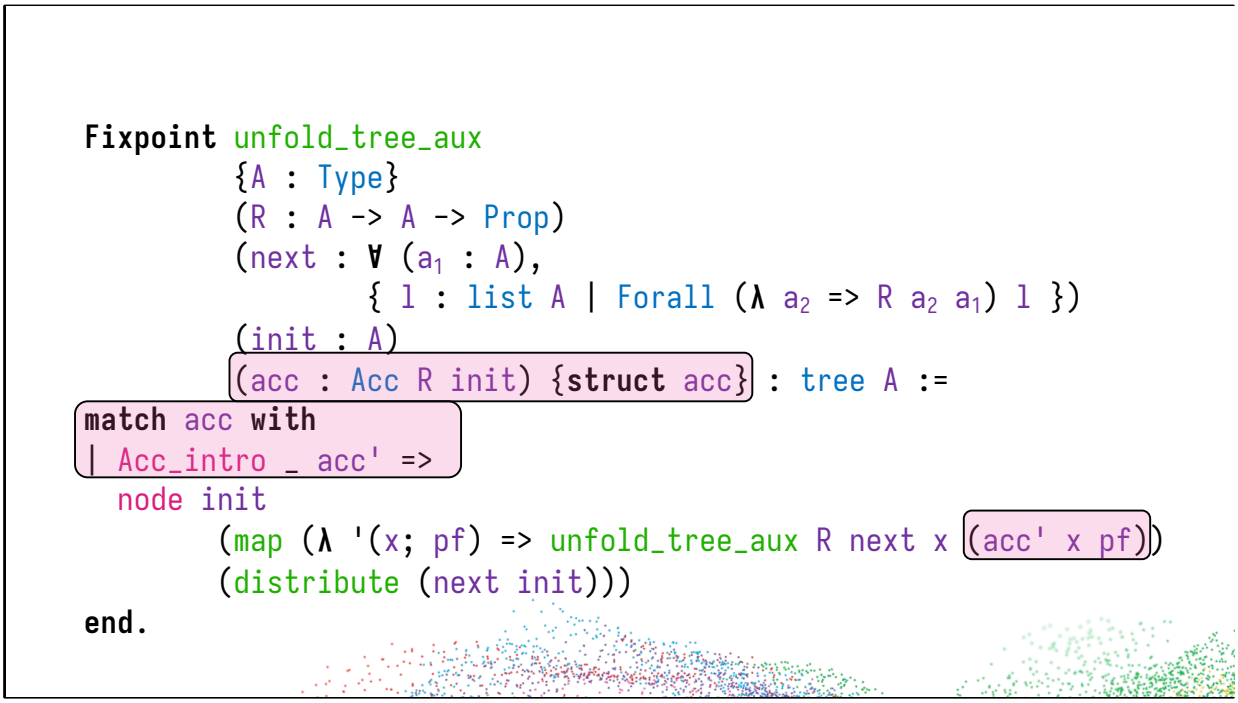
<click> Or take the strict subset relation on sets. There is no strict subset of empty set, so no infinite chain.

<click> Now, we have a game state type A, and possible game states of that type, a1, a2, and such. We do not know yet that the relation R on game states is well-founded. We need to prove that. I won't show you how to write that spec or how to prove that, the constructive formulation is a bit different than the one I present here, but I will show you how to use that proof to do general recursion.



We will use it by breaking our unfold function in half, into an entry point function and an auxiliary function.

Here we have an entry point function, that
<click> requires the relation R to be well-founded, and then uses that well-foundedness to generate an accessibility proof...



We also have an auxiliary function that takes that accessibility proof, <click> and uses it to perform recursion. I don't have time to show the definition of accessibility here today, but you can think of it this way: as long as we have a value that is "less than" what we started with, we can use that value in a recursive call. Since there can be no infinite descending chain, this function has to terminate.

This idea is the essence of all methods to perform general recursion in Rocq. Methods like Program Fixpoint or Equations also use accessibility under the hood.

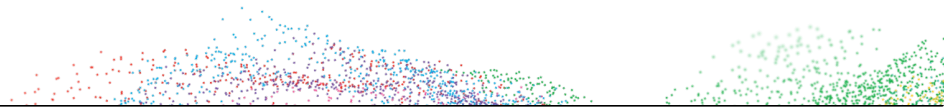
And thanks to this we now have an unfold function for **inductive** game trees as well. But whenever we want to use them for a game, we have to define a relation on game states for that game, and prove that the relation that describes the moves of the games is well-founded.

Lessons to learn:

- General recursion on inductive types is well-understood, but it can be a bit **labor intensive**.

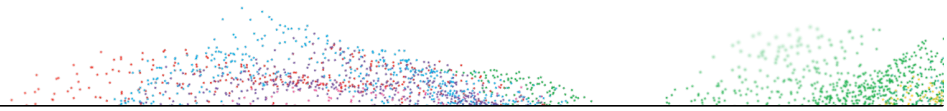
Lessons we do not have time for today, but are in the paper:

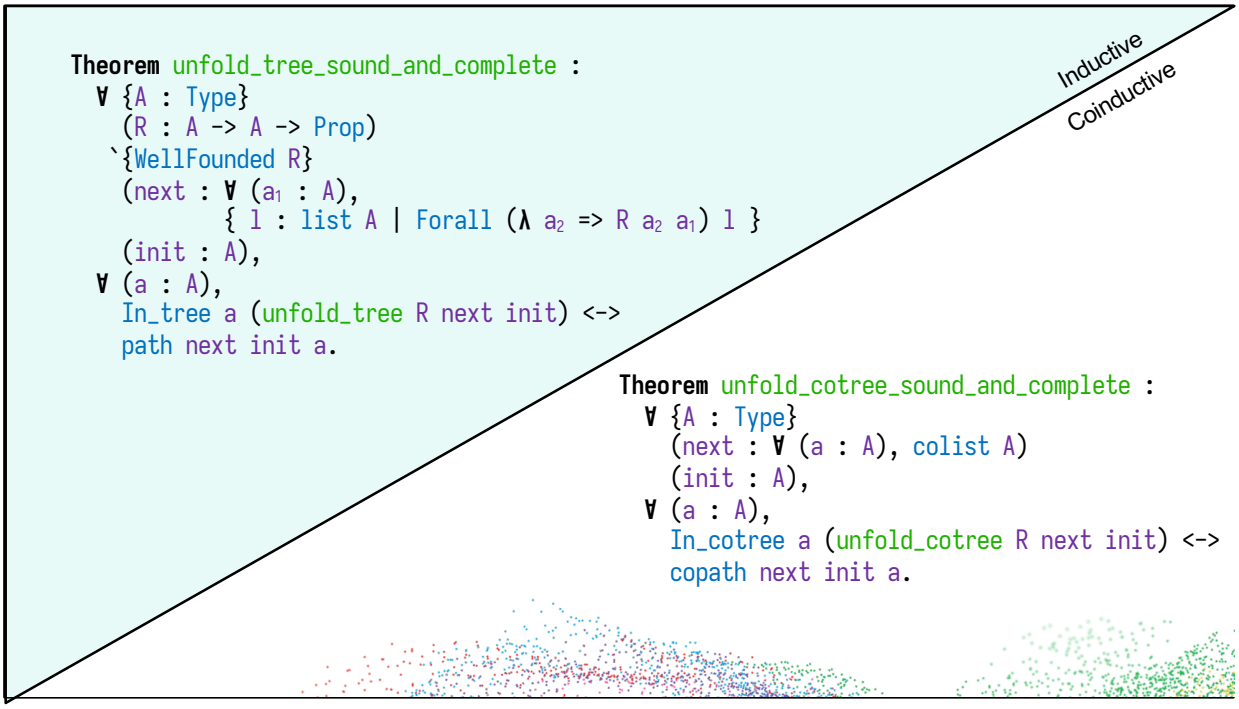
- Getting around the termination checker can require carrying around **extra bits of proof**, which forces us to make our verification more **intrinsic**.



How do we know we are generating the right trees?

- A Rocq proof of **"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.
- A Rocq proof of **"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.
- **Proved for both inductive and coinductive trees!**





This is what the theorem statements for soundness and completeness look like in Rocq, for both inductive and coinductive game trees.

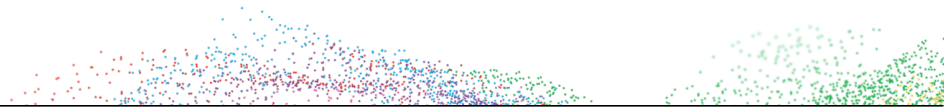
In both, we say that a game state is in the unfolded game tree if and only if there is a sequence of moves (in other words, a “path”) from the initial state to that state.

In the paper, I talk about useful tricks for proving these theorems in Rocq and how to avoid certain pitfalls.

Application: SAT solver

```
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.

Definition find_sat (f : formula) : option (list (name * bool)) :=
  fold_tree
    (λ (g : game) (l : list (option (list (name * bool)))) =>
       match eval (apply_assignments f g) with
       | Some true => Some g
       | _ => hd_error (somes l)
       end)
    (unfold_tree (later f) (sat_next_intrinsic f) []).
```

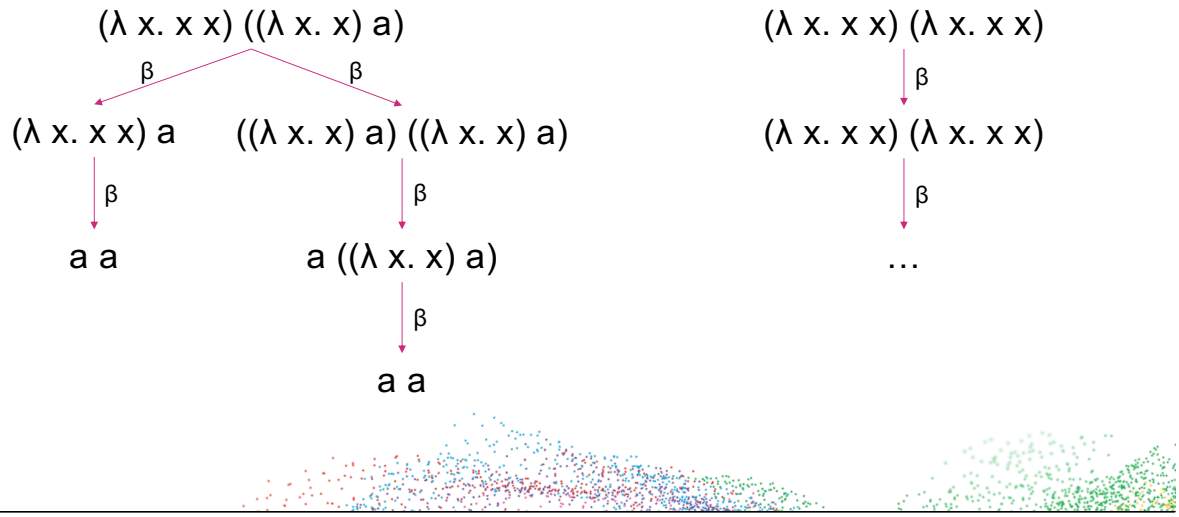


Also in the paper, we show example applications of unfolding game trees. The first example we present is a tic-tac-toe game, with an unbeatable AI thanks to the game tree we build through unfolding.

The second example is a very primitive SAT solver that tries to find a model for a formula by trying both true and false for every variable.
Once we prove that such an assignment actually decreases the number of free variables in a formula, …
<click> we can then fold that tree and find a set of assignments that make the formula true. And that is all we need to have a SAT solver.

In the paper, we also have a short discussion of efficiency concerns and more clever search methods over the game trees we build.

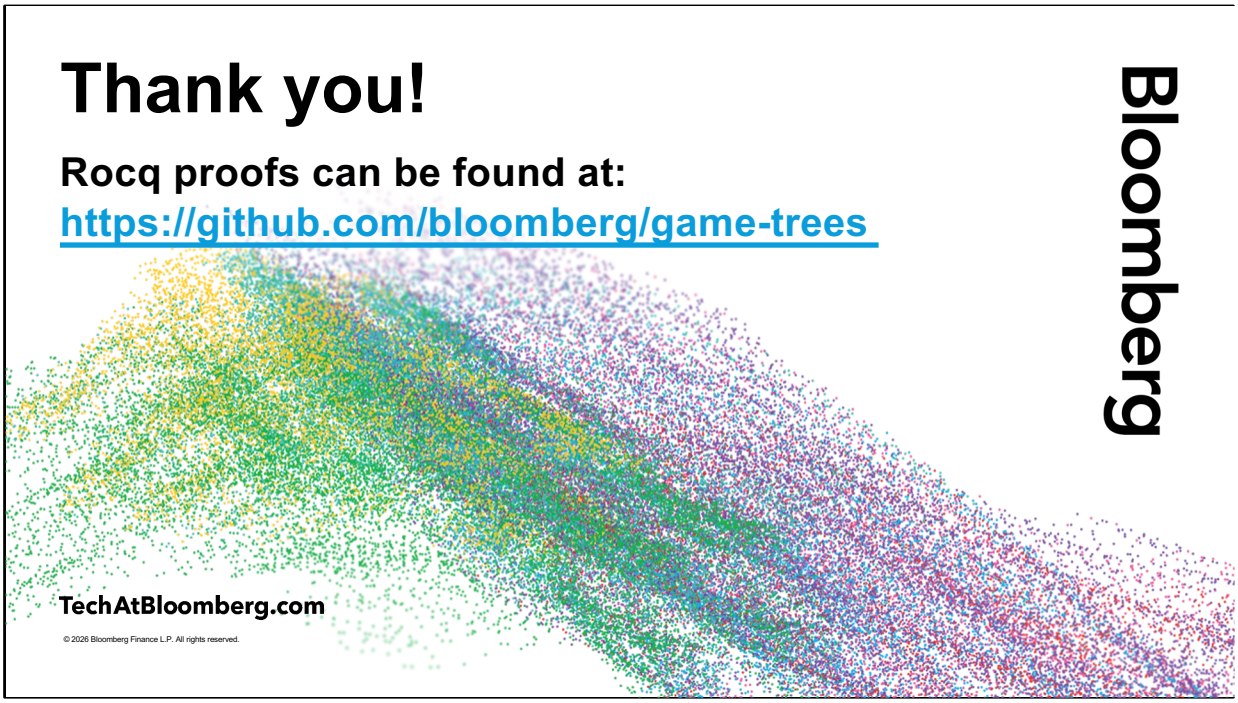
Possible application: λ-calculus



I want to end the talk with another possible application. Since this is mostly a PL audience, imagine evaluating lambda calculus terms as a game, where your initial state is the expression you want to evaluate, and reductions are the moves in the game. For lambda calculus, this will be an infinite game, since you can have infinite sequences of reductions, so you will want **coinductive** game trees.

But if we were talking about simply typed lambda calculus, we may have wanted to use **inductive** game trees, because we know simply typed lambda calculus is strongly normalizing. However, to unfold the inductive game tree, you'd have to provide a well-founded relation, and that is equivalent to the strong normalization proof!

Why might you want to reason about lambda calculi with game trees? Game trees let you look at different paths of evaluation at the same time. And if your next states function contains all possible reductions, you can reason about what reductions are possible and what are not possible, at the same time.



Thank you so much for listening!

If anyone is interested, the repo of the Rocq proofs is open sourced! Check out the Bloomberg organization on GitHub.