A Rose Tree is Blooming
(Proof Pearl)

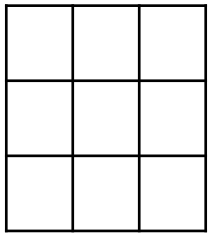
Bloomberg

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

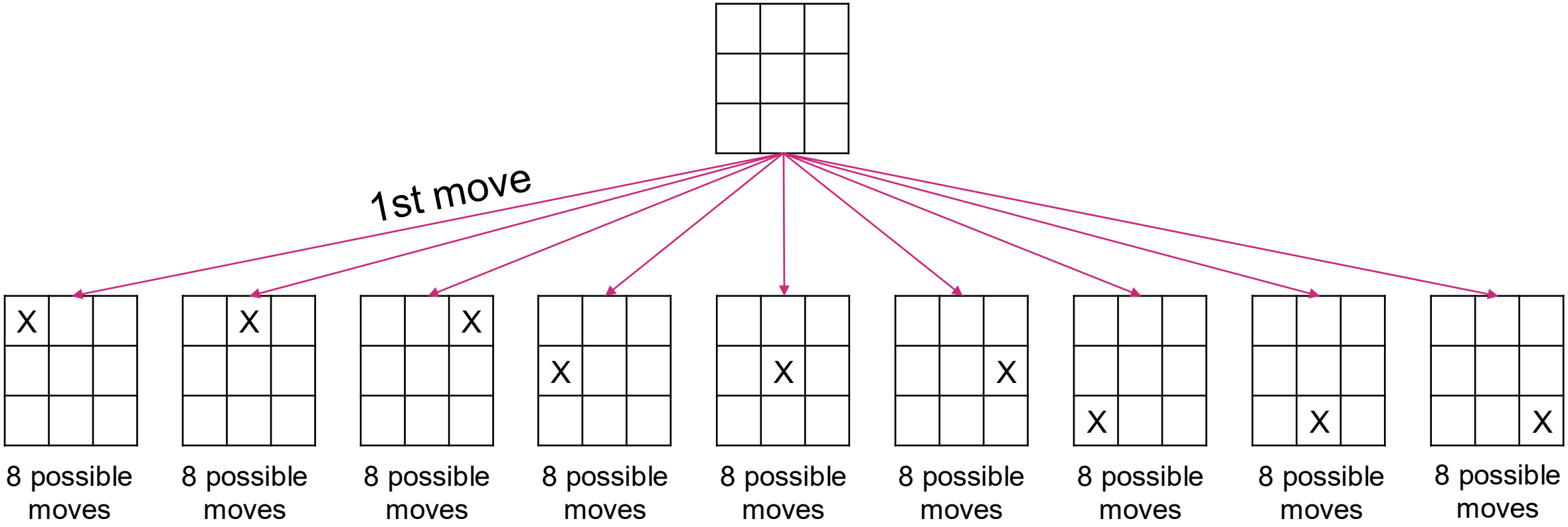
TechAtBloomberg.com

A game tree

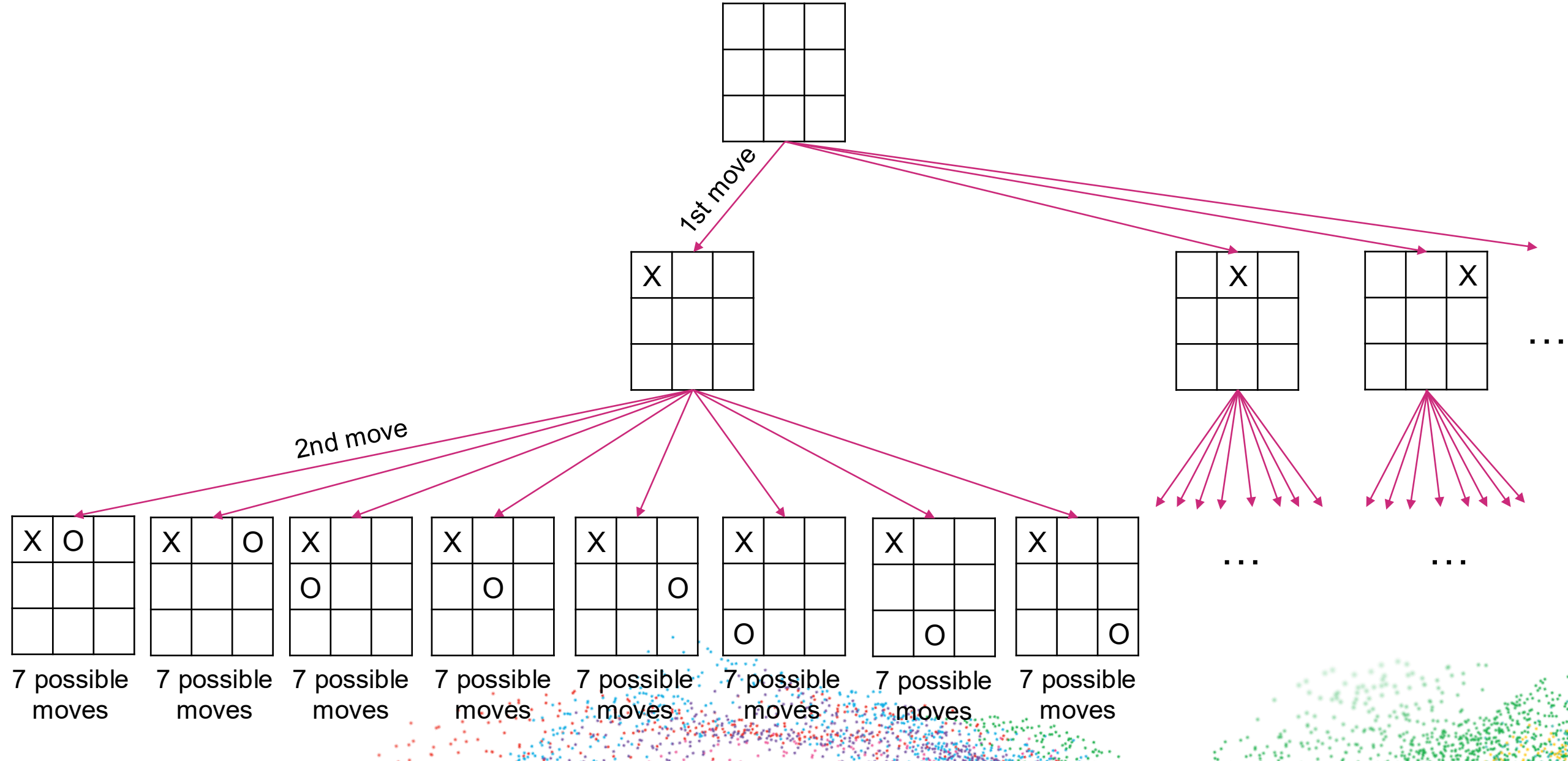


9 possible moves

A game tree


1st move
8 possible moves
8 possible moves
8 possible moves
8 possible moves
8 possible moves
8 possible moves
8 possible moves
8 possible moves
8 possible moves

A game tree

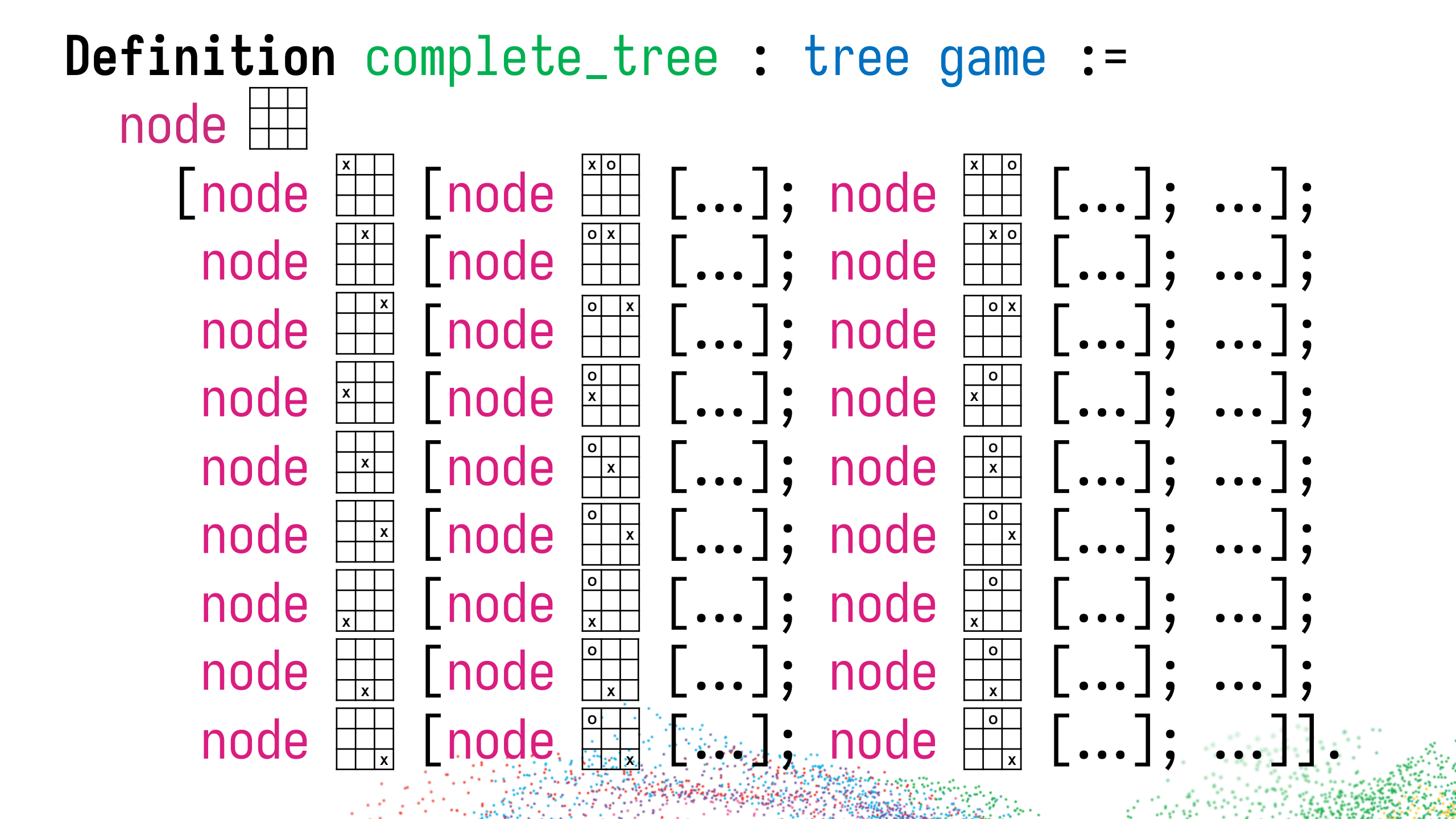

1st move
2nd move
X
X
X
...
...
...
X O
X O
X O
X O
X O
X O
X O
7 possible moves
7 possible moves
7 possible moves
7 possible moves
7 possible moves
7 possible moves
7 possible moves
7 possible moves

```
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.
```

(written in the Rocq Prover, formerly known as the Coq proof assistant)

```
Definition complete_tree : tree game :=
  node
    [node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...];
     node  [node  [...]; node  [...]; ...]].
```



```
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.
```

Does this even necessarily terminate?

```
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.
```

```
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.
```

```
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.
```

```
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.

```
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.
```

```
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.
```

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.
```

```
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.
```

Well-foundedness of a relation

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

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

$$? \ldots a_6 \ \mathbf{R} \ a_5 \ \mathbf{R} \ a_4 \ \mathbf{R} \ a_3 \ \mathbf{R} \ a_2 \ \mathbf{R} \ a_2 \ \mathbf{R} \ a_1$$

```
Definition unfold_tree
           {A : Type}
           (R : A -> A -> Prop)
           `{WellFounded R}
           (next : ∀ (a₁ : A),
                   { l : list A | Forall (λ a₂ => R a₂ a₁) l })
           (init : A) : tree A :=
 unfold_tree_aux R next init (wellfounded init).
```

```
Fixpoint unfold_tree_aux
         {A : Type}
         (R : A -> A -> Prop)
         (next : ∀ (a₁ : A),
                 { l : list A | Forall (λ a₂ => R a₂ a₁) l })
         (init : A)
         (acc : Acc R init) {struct acc} : tree A :=
match acc with
| Acc_intro _ acc' =>
  node init
       (map (λ '(x; pf) => unfold_tree_aux R next x (acc' x pf))
       (distribute (next init)))
end.
```

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!**

```
Theorem unfold_tree_sound_and_complete :
  ∀ {A : Type}
    (R : A -> A -> Prop)
   `{WellFounded R}
    (next : ∀ (a₁ : A),
            { l : list A | Forall (λ a₂ => R a₂ a₁) l }
    (init : A),
  ∀ (a : A),
    In_tree a (unfold_tree R next init) <->
    path next init a.
```

Inductive

Coinductive

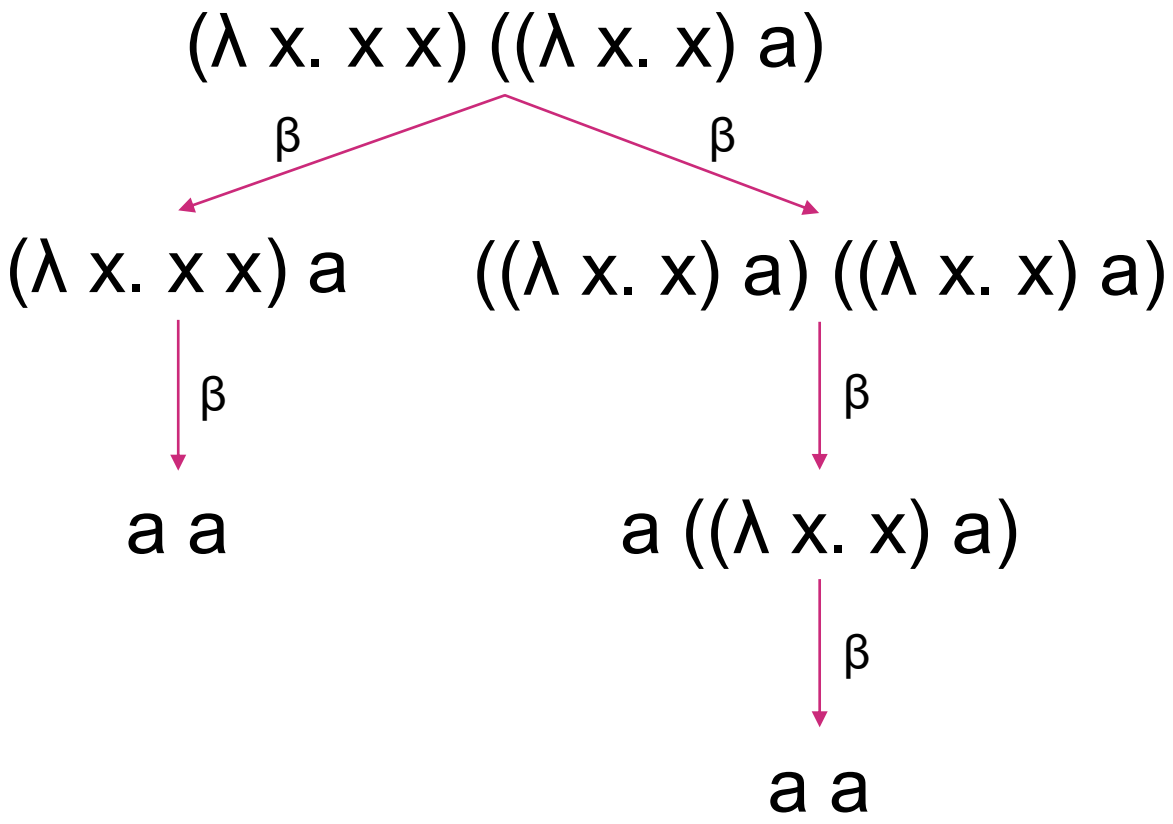
```
Theorem unfold_cotree_sound_and_complete :
  ∀ {A : Type}
    (next : ∀ (a : A), colist A)
    (init : A),
  ∀ (a : A),
    In_cotree a (unfold_cotree R next init) <->
    copath next init a.
```

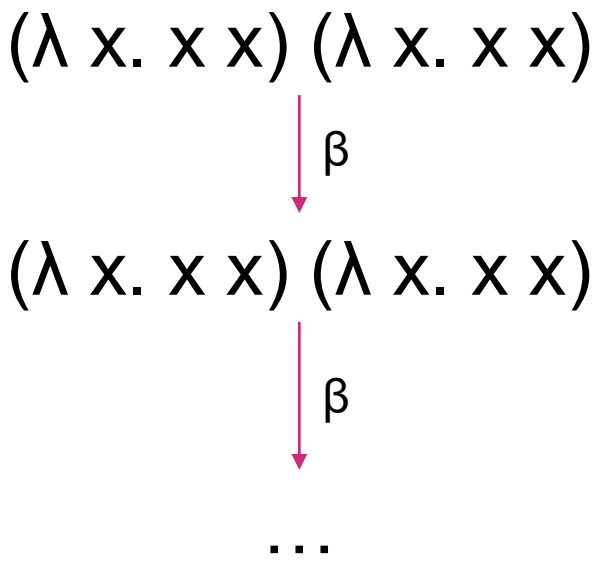
Application: SAT solver

```
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) []).
```

Possible application: λ-calculus

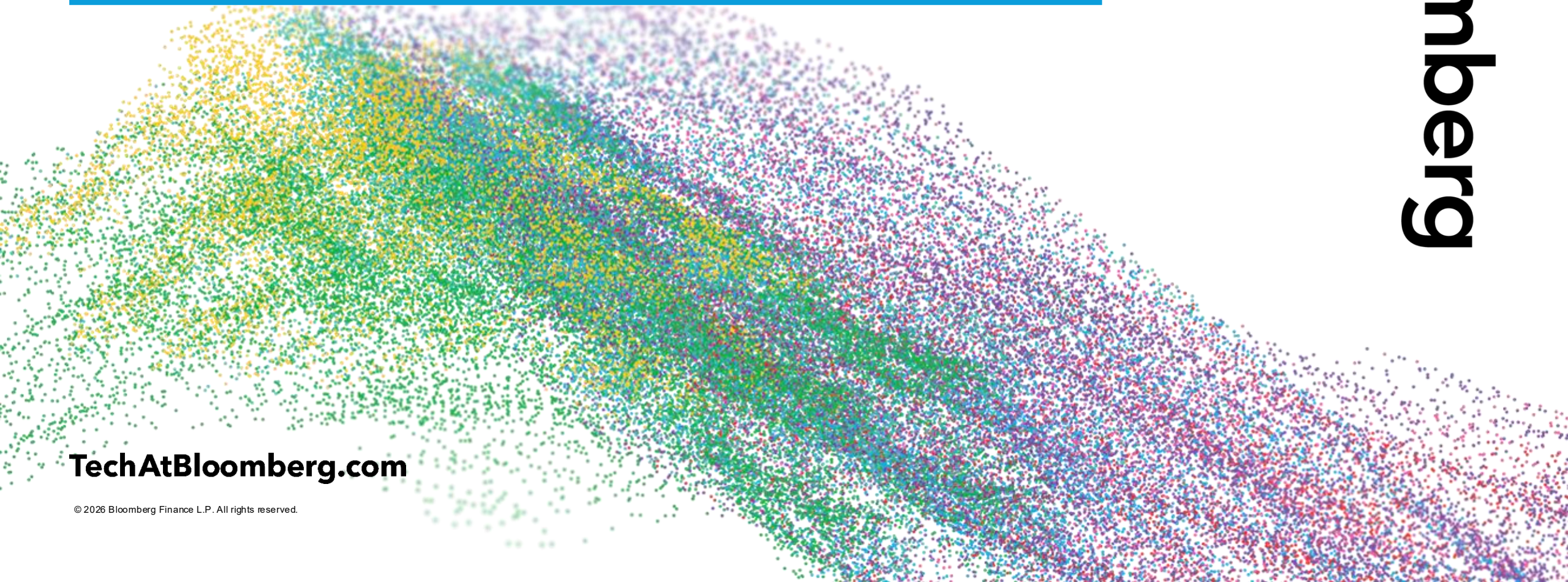




Thank you!

Rocq proofs can be found at:

https://github.com/bloomberg/game-trees



Bloomberg

TechAtBloomberg.com