Formalizing Possibly Infinite Trees of Finite Degree

König's Lemma for Chase Termination

Lukas Gerlach

Knowledge-Based Systems Group, TU Dresden, Germany

02.10.2025

Technische Universität Dresden

International Center for Computational Logic

Background / Short Motivation 💡

My work revolves around the chase algorithm on existential rules.

See https://dmfa.dev/lean

Background / Short Motivation 💡

My work revolves around the chase algorithm on existential rules.

See https://dmfa.dev/lean

$$\boxed{R(x, y) \to \exists z. R(y, z) \lor R(y, x)}$$

Background / Short Motivation 💡

My work revolves around the chase algorithm on existential rules.

See https://dmfa.dev/lean

$$\boxed{R(x, y) \to \exists z.R(y, z) \lor R(y, x)}$$

With the fact $R(a, b)$, we can obtain different `ChaseBranches`.

Background / Short Motivation 💡

My work revolves around the chase algorithm on existential rules.

See https://dmfa.dev/lean

$$\boxed{R(x, y) \to \exists z. R(y, z) \lor R(y, x)}$$

With the fact $R(a, b)$, we can obtain different `ChaseBranches`.

$$\boxed{R(a, b)}$$

Background / Short Motivation 💡

My work revolves around the chase algorithm on existential rules.

See https://dmfa.dev/lean

$$\boxed{R(x, y) \to \exists z.\ R(y, z) \lor R(y, x)}$$

With the fact $R(a, b)$, we can obtain different `ChaseBranches`.

$$\boxed{R(a, b)} \text{ --- } \boxed{R(b, n_1)}$$

Background / Short Motivation 💡

My work revolves around the chase algorithm on existential rules.

See https://dmfa.dev/lean

$$\boxed{R(x, y) \to \exists z. R(y, z) \vee R(y, x)}$$

With the fact $R(a, b)$, we can obtain different `ChaseBranches`.

$$\boxed{R(a, b)} \text{ --- } \boxed{R(b, n_1)} \text{ --- } \boxed{R(n_1, b)}$$

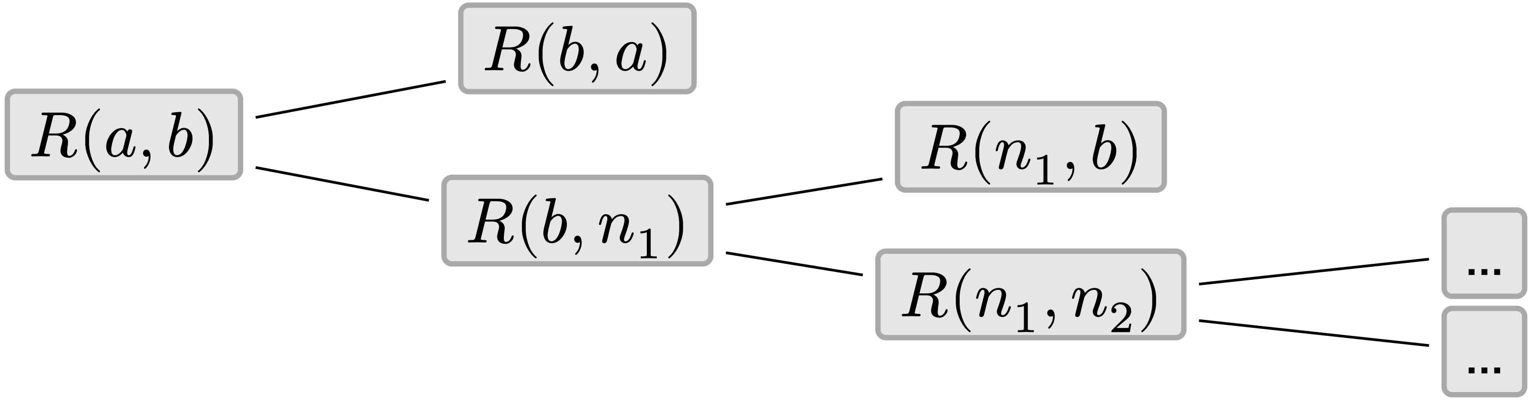
Background / Short Motivation 💡

My work revolves around the chase algorithm on existential rules.

See https://dmfa.dev/lean

$$\boxed{R(x, y) \to \exists z.R(y, z) \lor R(y, x)}$$

We can generalize this into an infinite `ChaseTree`.



What about inductive trees? 🌲

What about inductive trees? 🌲

```
inductive BinaryTree a with
```

What about inductive trees? 🌲

```
inductive BinaryTree a with
| leaf a -> BinaryTree a
```

What about inductive trees? 🌲

```
inductive BinaryTree a with
| leaf a -> BinaryTree a
| inner a -> BinaryTree a -> BinaryTree a -> BinaryTree a
```

What about inductive trees? 🌲

```
inductive BinaryTree a with
| leaf a -> BinaryTree a
| inner a -> BinaryTree a -> BinaryTree a -> BinaryTree a
```

Inductive types express the least fixed point, i.e. the minimal set of trees obtainable from the two constructors.

What about inductive trees? 🌲

```
inductive BinaryTree a with
| leaf a -> BinaryTree a
| inner a -> BinaryTree a -> BinaryTree a -> BinaryTree a
```

Inductive types express the least fixed point, i.e. the minimal set of trees obtainable from the two constructors. We want the greatest fixed point instead, i.e. a coinductive definition!

Plan of Attack ⚔️

1.
2.
3.
4.
5.

https://dmfa.dev/trees

Plan of Attack ⚔️

1. Take a hint from infinite lists.
2.
3.
4.
5.

https://dmfa.dev/trees

Plan of Attack ⚔️

1. Take a hint from infinite lists.
2. Define possibly infinite lists.
3.
4.
5.

https://dmfa.dev/trees

Plan of Attack ⚔️

1. Take a hint from infinite lists.
2. Define possibly infinite lists.
3. Define possibly infinite trees with finite degree.
4.
5.

https://dmfa.dev/trees

Plan of Attack ⚔️

1. Take a hint from infinite lists.
2. Define possibly infinite lists.
3. Define possibly infinite trees with finite degree.
4. Define branches in trees.
5.

https://dmfa.dev/trees

Plan of Attack ⚔️

1. Take a hint from infinite lists.
2. Define possibly infinite lists.
3. Define possibly infinite trees with finite degree.
4. Define branches in trees.
5. Prove König's Lemma.

https://dmfa.dev/trees

Plan of Attack ⚔️

1. Take a hint from infinite lists.
2. Define possibly infinite lists.
3. Define possibly infinite trees with finite degree.
4. Define branches in trees.
5. Prove König's Lemma.

König's Lemma: (Special Case for Trees)

Plan of Attack ⚔️

1. Take a hint from infinite lists.
2. Define possibly infinite lists.
3. Define possibly infinite trees with finite degree.
4. Define branches in trees.
5. Prove König's Lemma.

König's Lemma: (Special Case for Trees)

If each branch of a tree with finite degree is finite,

Plan of Attack ⚔️

1. Take a hint from infinite lists.
2. Define possibly infinite lists.
3. Define possibly infinite trees with finite degree.
4. Define branches in trees.
5. Prove König's Lemma.

König's Lemma: (Special Case for Trees)
If each branch of a tree with finite degree is finite,
then there are only finitely many branches.

Infinite Lists aka. Stream' 🏄

```
def Stream' (α : Type u) := Nat → α -- From Mathlib
-- We call the same thing InfiniteList instead.
```

Infinite Lists aka. Stream' 🏄

```
def Stream' (α : Type u) := Nat → α -- From Mathlib
-- We call the same thing InfiniteList instead.
```

How to make this possibly infinite?

Infinite Lists aka. Stream' 🏄

```
def Stream' (α : Type u) := Nat → α -- From Mathlib
-- We call the same thing InfiniteList instead.
```

How to make this possibly infinite? How about `InfiniteList (Option α)`?

Infinite Lists aka. Stream' 🏄

```
def Stream' (α : Type u) := Nat → α -- From Mathlib
-- We call the same thing InfiniteList instead.
```

How to make this possibly infinite? How about `InfiniteList (Option α)`?

```
structure PossiblyInfiniteList (α : Type u) where
  infinite_list : InfiniteList (Option α)
  no_holes : ∀ n : Nat, infinite_list n ≠ none ->
             ∀ m : Fin n, infinite_list m ≠ none
```

Infinite Lists aka. Stream' 🏄

```
def Stream' (α : Type u) := Nat → α -- From Mathlib
-- We call the same thing InfiniteList instead.
```

How to make this possibly infinite? How about `InfiniteList (Option α)`?

```
structure PossiblyInfiniteList (α : Type u) where
  infinite_list : InfiniteList (Option α)
  no_holes : ∀ n : Nat, infinite_list n ≠ none ->
             ∀ m : Fin n, infinite_list m ≠ none
```

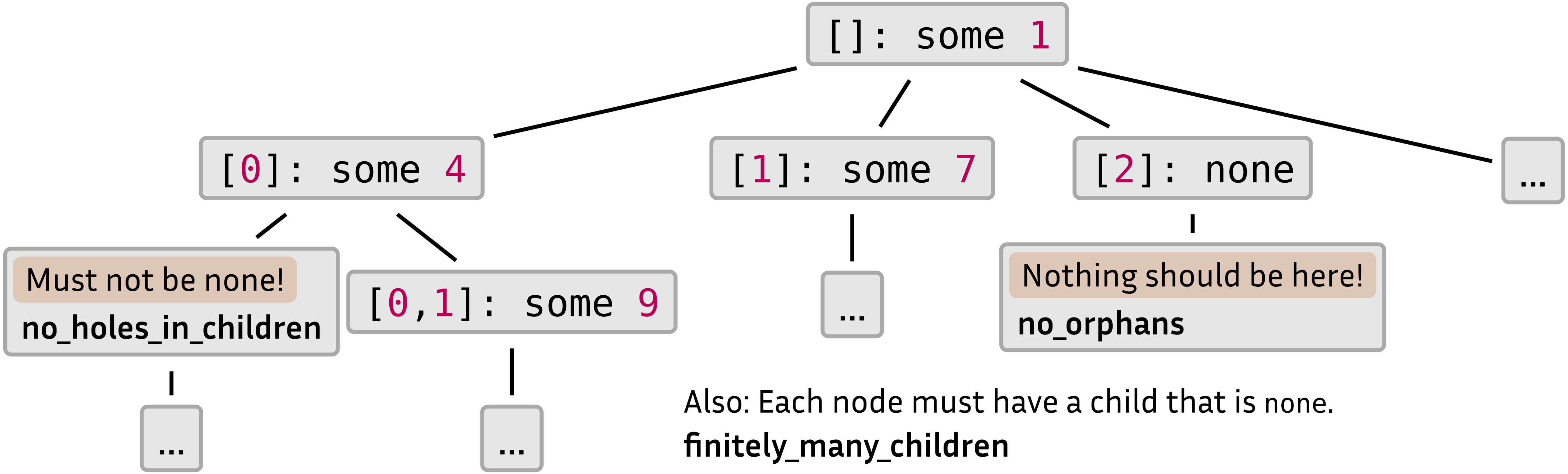
Now we only need to turn the `InfiniteList` into an `InfiniteTree`. How?

From List to Tree - Change the Address!

```
-- List addresses are `Nat`. Tree addresses are `List Nat`!
def FiniteDegreeTreeStub (a : Type u) := List Nat -> Option a
```

From List to Tree - Change the Address! 📫

```
-- List addresses are `Nat`. Tree addresses are `List Nat`!
def FiniteDegreeTreeStub (a : Type u) := List Nat -> Option a
```



Also: Each node must have a child that is none.
finitely_many_children

Defining Branches 🪾

Defining Branches 🌳

```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
```

Defining Branches 🪾

```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
def branches (t : PIT α) : Set (PIL α) := t.branches_through []
```

Defining Branches 🌳

```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
def branches (t : PIT α) : Set (PIL α) := t.branches_through []
def branches_through (t : PIT α) (n : List Nat) : Set (PIL α) :=
  (t.addresses_through n).map t.branch_for_address
```

Defining Branches 🌳

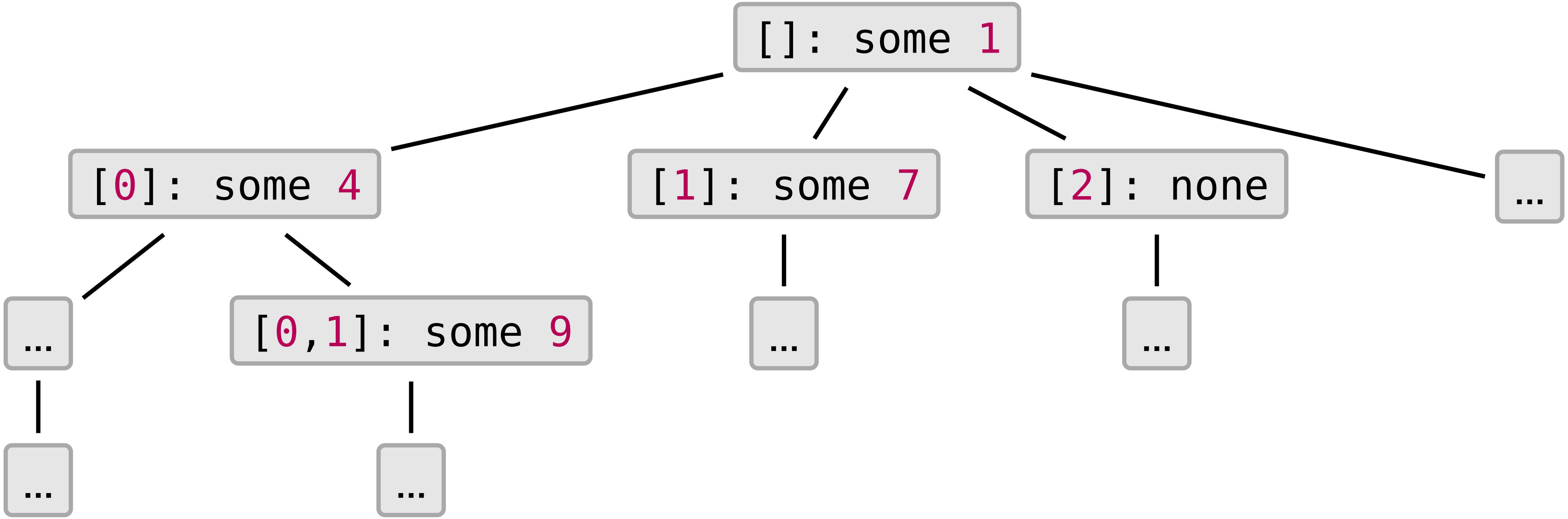
```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
def branches (t : PIT α) : Set (PIL α) := t.branches_through []
def branches_through (t : PIT α) (n : List Nat) : Set (PIL α) :=
  (t.addresses_through n).map t.branch_for_address
def addresses_through (t : PIT α) (n : List Nat) : Set (IL Nat) :=
  fun ns => ns ∈ ns.take node.length = node
```

Defining Branches

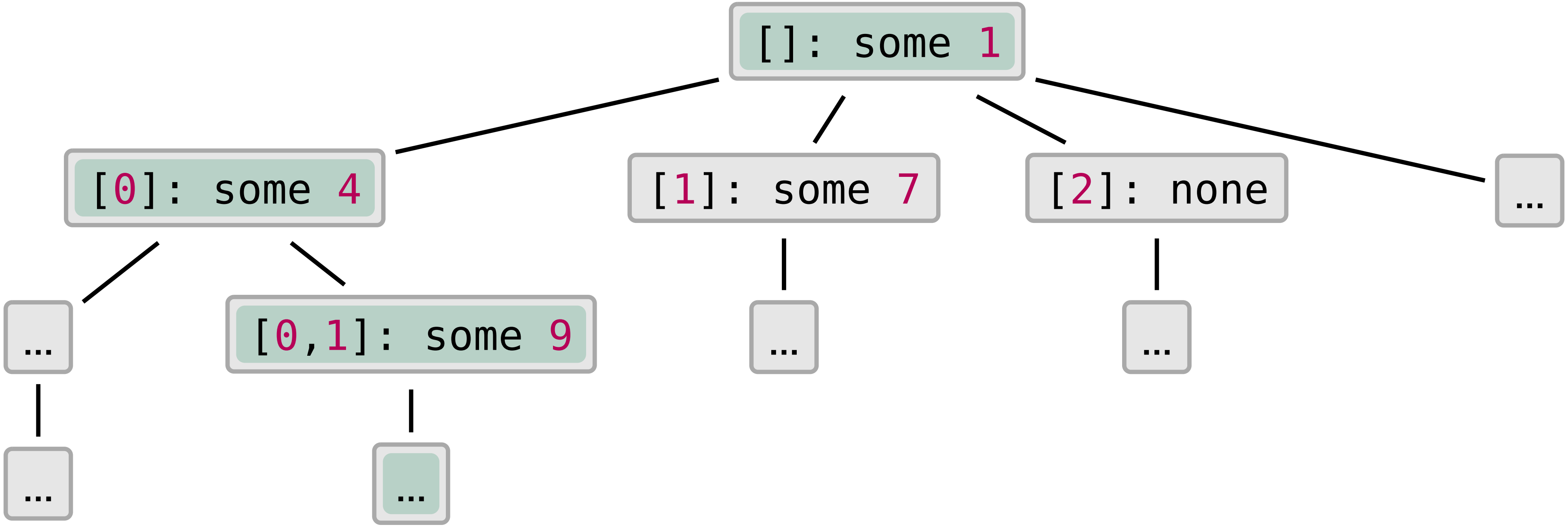
```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
def branches (t : PIT α) : Set (PIL α) := t.branches_through []
def branches_through (t : PIT α) (n : List Nat) : Set (PIL α) :=
  (t.addresses_through n).map t.branch_for_address
def addresses_through (t : PIT α) (n : List Nat) : Set (IL Nat) :=
  fun ns => ns ∈ ns.take node.length = node
```

Is that it?

Defining Branches 🌳 (2)

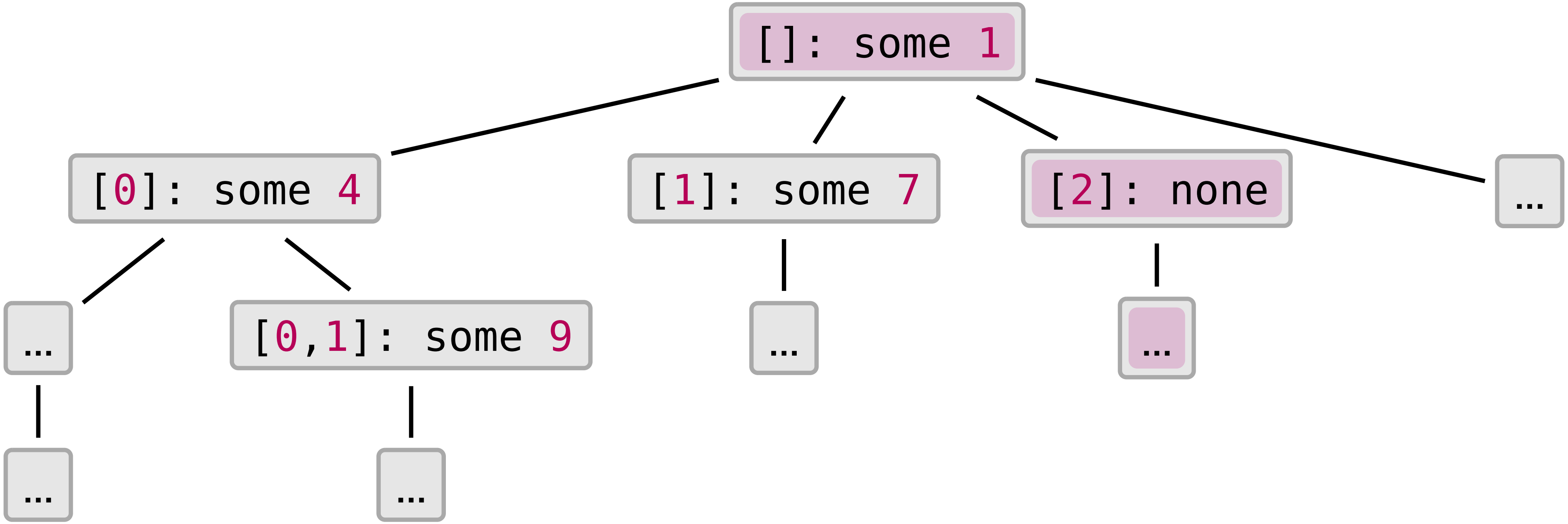


Defining Branches 🥀 (2)



Should the green be a valid branch address? ✋

Defining Branches 🥀 (2)



Should the pink be a valid branch address? ✋

Defining Branches 🪾 (3)

```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
def branches (t : PIT α) : Set (PIL α) := t.branches_through []
def branches_through (t : PIT α) (n : List Nat) : Set (PIL α) :=
  (t.addresses_through n).map t.branch_for_address
def addresses_through (t : PIT α) (n : List Nat) : Set (IL Nat) :=
  fun ns => ns ∈ ns.take n.length = n ∧ t.address_is_maximal ns
```

Defining Branches 🪾 (3)

```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
def branches (t : PIT α) : Set (PIL α) := t.branches_through []
def branches_through (t : PIT α) (n : List Nat) : Set (PIL α) :=
  (t.addresses_through n).map t.branch_for_address
def addresses_through (t : PIT α) (n : List Nat) : Set (IL Nat) :=
  fun ns => ns ∈ ns.take n.length = n ∧ t.address_is_maximal ns
def address_is_maximal (t : PIT α) (ns : IL Nat) : Prop :=
  -- If the branch is not infinite, it should end in a leaf
  ∀ n, t (ns.take (n+1)) = none -> t (0 :: (ns.take n)) = none
```

Defining Branches 🪾 (3)

```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
def branches (t : PIT α) : Set (PIL α) := t.branches_through []
def branches_through (t : PIT α) (n : List Nat) : Set (PIL α) :=
  (t.addresses_through n).map t.branch_for_address
def addresses_through (t : PIT α) (n : List Nat) : Set (IL Nat) :=
  fun ns => ns ∈ ns.take n.length = n ∧ t.address_is_maximal ns
def address_is_maximal (t : PIT α) (ns : IL Nat) : Prop :=
  -- If the branch is not infinite, it should end in a leaf
  ∀ n, t (ns.take (n+1)) = none -> t (0 :: (ns.take n)) = none
```

Defining Branches 🪾 (3)

```
-- Start from the end; PIT = PossiblyInfiniteTree; PIL = PIList
def branches (t : PIT α) : Set (PIL α) := t.branches_through []
def branches_through (t : PIT α) (n : List Nat) : Set (PIL α) :=
  (t.addresses_through n).map t.branch_for_address
def addresses_through (t : PIT α) (n : List Nat) : Set (IL Nat) :=
  fun ns => ns ∈ ns.take n.length = n ∧ t.address_is_maximal ns
def address_is_maximal (t : PIT α) (ns : IL Nat) : Prop :=
  -- If the branch is not infinite, it should end in a leaf
  ∀ n, t (ns.take (n+1)) = none -> t (0 :: (ns.take n)) = none
```

📢 This also ensures that every `ChaseTree` branch is a proper `ChaseBranch`.

Proving König's Lemma 👑

König's Lemma: (Special Case for Trees)
If each branch of a tree with finite degree is finite,
then there are only finitely many branches.

Proving König's Lemma 👑

```
theorem branches_finite_of_each_branch_finite (t : FinDegTree α) :
  (∀ b, b ∈ t.branches -> ∃ i, b i = none) -> t.branches.fin
```

Proving König's Lemma 👑

```
theorem branches_finite_of_each_branch_finite (t : FinDegTree α) :
  (∀ b, b ∈ t.branches -> ∃ i, b i = none) -> t.branches.fin
-- towards contradiction, "Classical.choose" an infinite branch
have : ∃ (ns : IL Nat), ∀ i, ¬ t.branches_through (ns.take i).fin
```

Proving König’s Lemma 👑

```
theorem branches_finite_of_each_branch_finite (t : FinDegTree α) :
  (∀ b, b ∈ t.branches -> ∃ i, b i = none) -> t.branches.fin
-- towards contradiction, "Classical.choose" an infinite branch
have : ∃ (ns : IL Nat), ∀ i, ¬ t.branches_through (ns.take i).fin
let ns := fun n => (inf_node contra n.succ).val.head
```

Proving König's Lemma 👑

```
theorem branches_finite_of_each_branch_finite (t : FinDegTree α) :
  (∀ b, b ∈ t.branches -> ∃ i, b i = none) -> t.branches.fin
-- towards contradiction, "Classical.choose" an infinite branch
have : ∃ (ns : IL Nat), ∀ i, ¬ t.branches_through (ns.take i).fin
let ns := fun n => (inf_node contra n.succ).val.head
noncomputable def inf_node {t} (n_fin : ¬ t.branches.fin) (d) :
  { n : List Nat // n.length = d ∧ ¬ (t.branches_through n).fin }
```

Proving König’s Lemma 👑

```
theorem branches_finite_of_each_branch_finite (t : FinDegTree α) :
  (∀ b, b ∈ t.branches -> ∃ i, b i = none) -> t.branches.fin
-- towards contradiction, "Classical.choose" an infinite branch
have : ∃ (ns : IL Nat), ∀ i, ¬ t.branches_through (ns.take i).fin
let ns := fun n => (inf_node contra n.succ).val.head
noncomputable def inf_node {t} (n_fin : ¬ t.branches.fin) (d) :
  { n : List Nat // n.length = d ∧ ¬ (t.branches_through n).fin }
theorem inf_node_extends_prev {t} (n_fin : ¬ t.branches.fin) (d) :
  (inf_node n_fin d.succ).val =
    (inf_node n_fin d.succ).val.head :: (inf_node n_fin d).val
```

Proving König's Lemma 👑

```
theorem branches_finite_of_each_branch_finite (t : FinDegTree α) :
  (∀ b, b ∈ t.branches -> ∃ i, b i = none) -> t.branches.fin
-- towards contradiction, "Classical.choose" an infinite branch
have : ∃ (ns : IL Nat), ∀ i, ¬ t.branches_through (ns.take i).fin
let ns := fun n => (inf_node contra n.succ).val.head
noncomputable def inf_node {t} (n_fin : ¬ t.branches.fin) (d) :
  { n : List Nat // n.length = d ∧ ¬ (t.branches_through n).fin }
theorem inf_node_extends_prev {t} (n_fin : ¬ t.branches.fin) (d) :
  (inf_node n_fin d.succ).val =
    (inf_node n_fin d.succ).val.head :: (inf_node n_fin d).val
```

📢 If every branch in a `ChaseTree` is finite, so is the whole `ChaseTree`.

This talk is finite and we've reached its end 🏁

Thank you so much for having me! 🤓

I hope you can get all your `goals accomplished` 🎉

Feel free to reach out in Lean Zulipchat or via mail:

`lukas.gerlach@tu-dresden.de`

`hi@monsterkrampe.dev`

⭐ Check out https://dmfa.dev/lean and https://dmfa.dev/trees ⭐