A Formally Verified Proof of Kruskal's Tree Theorem in Lean

Minchao Wu
Advisor: Jeremy Avigad

Abstract

We describe a formalization of Kruskal's tree theorem in a new proof assistant called Lean. The formalization follows the classical paper proof given by Nash-Williams in which the minimal bad sequence argument plays a central role in proving the theorem. We formalize this argument as an independent module so that it can be applied to later proofs. Along the way, Dickson's lemma and Higman's lemma are formalized in a manner similar to Nash-Williams' original proof.

Contents

1 Introduction

Kruskal's tree theorem is a famous theorem in combinatorics. The theorem says that finite trees are well quasi-oredered under homeomorphic embedding. A set A is well quasi-ordered by a relation $\preceq$ if $\preceq$ is reflexive and transitive, and for every infinite sequence f of elements of A, there exist $i, j \in \mathbb{N}$ such that $i < j$ and $f(i) \preceq f(j)$. Intuitively, a finite tree T is embeddable into another finite tree T' via a homeomorphism h, if for every node $n \in T$, the image under h of the successors of n are distinct successors of $h(n)$. In the term rewriting literature, the tree theorem turns out to be a powerful tool for proving termination. It implies that if certain orderings defined recursively on terms satisfy some simplification property, then the well-foundedness of these orderings can be obtained [14] [12]. These well-founded orderings in turn establish the termination of systems of rewrite rules and the correctness of algorithms like Knuth-Bendix completion procedures [7].

From a proof-theoretic perspective, Kruskal's tree theorem is significant in that it is a natural statement unprovable in relatively strong logical theories, though it is a theorem about finite objects. Let T be a formal theory that contains sufficient arithmetic to make statements about ordinal notations. The proof-theoretic ordinal of T is the smallest recursive ordinal α such that the transfinite induction up to α is not provable in T. Let σ be a mathematical statement. It is clear that if T proves that σ implies the well-foundedness of some ordinal α, and T does not prove the well-foundedness of some ordinal α, then σ is not provable in T. It is well-known that the proof-theoretic ordinal of certain strong systems of second-order arithmetic, such as the system ATR_0, is the Feferman-Schütte ordinal Γ_0. On the other hand, Harvey Friedman [4] shows that the system ACA_0 proves that Kruskal's theorem implies the well-foundedness of Γ_0. Since ACA_0 is a subsystem of ATR_0, this means that Kruskal's theorem is not provable in ATR_0. This result is interesting because ATR_0 is already able to formalize a large portion of ordinary mathematics, including the theory of continuous functions, the theory of countable fields, the topology of complete separable metric spaces, the structure theory of separable Banach spaces, Borel sets, analytic sets, etc. [5], while it fails to figure out the "truth" of a combinatorial statement about finite objects. More investigations of proof-theoretic consequences of Kruskal's tree theorem and its variants can be found in Simpson [13], Okada and Takeuti [11].

Kruskal's tree theorem, which is practically useful and theoretically interesting, was conjectured by Andrew Vázsonyi and first proved by Joseph Kruskal [8]. Nash-Williams gives an elegant and classical proof in [9]. The proof is classical in the sense that it uses axiom of choice and proof by con-

tradiction. The minimal bad sequence argument has been established in the same paper in order to obtain contradictions. The argument assumes the existence of a "bad" sequence. Then it follows that there is a bad sequence which is minimal in some sense. Using this minimal bad sequence we can construct a new bad sequence which is even smaller in the same sense. The existence of this new bad sequence then contradicts the minimality of the minimal bad sequence. The minimal bad sequence argument is applied to the proof of a version of Higman's lemma [6] given in the same paper, as well as the proof of the tree theorem itself. In this paper, we formalize a general abstraction of the argument so that a contradiction can be obtained as long as suitable instances of the assumptions are constructed.

Formalizations of Kruskal's tree theorem are not established until very recently. Previous formalizations of Kruskal's tree theorem are given by Sternagel [15] in 2013 and Dominique Larchey-Wendling[1] in 2015. The latter one is a formalization of an intuitionistic proof of the theorem in Coq [3], so it cannot really be compared to our formalization since it adopts a different proof strategy. The former one is a formalization of the classical proof in Isabelle/HOL [10]. According to [15], this one appears to be the first formalization of the theorem and contains about 2000 lines of code. Our formalization in Lean [2] uses a simpler inductive definition of finite trees and formalizes a more general abstraction of the minimal bad sequence argument. The argument is more general so that there is no need to repeat the constructions of the contradicting bad sequence in the proof. Besides, we formalize Higman's lemma in a form as it is in Nash-Williams' original proof, i.e., in terms of finite subsets, so as to distinguish the formalization from the above two where they have formalized its list version[2]. The resulting formalization is about 1000 lines.

[1] `https://members.loria.fr/DLarchey/files/Kruskal/index.html`

[2] However, the strategy for proving both versions are almost the same once we have established the minimal bad sequence argument, and the last step for the list version is even simpler.

2 Background

We begin by describing the basics of the Lean theorem prover and the WQO theory, focusing on the parts relevant to our work.

2.1 The Lean Theorem Prover

The Lean theorem prover is an open source interactive theorem prover developed by Microsoft Research. The latest version of Lean supports both constructive and classical reasoning in the calculus of inductive constructions (CIC) — a powerful variant of Martin-Löf type theory in terms of expressiveness. What makes the theory powerful is the existence of dependent types and inductive types. Dependent types, denoted as $\Pi x : A, B$, can be considered as the type of an indexed family of sets. An element of $\Pi x : A, B$ is a function f which, for each a in A, returns an element $f\ a$ of $B\ a$. If x occurs in B, then $\Pi x : A, B$ represents a family of types indexed by the elements in A. If x does not occur in B, then $\Pi x : A, B$ represents the function type $A \to B$. In this sense, Π generalizes the usual $\to$. The typing rules for Π are given as follows.

$$\frac{\Gamma \vdash A : Type \qquad \Gamma, x : A \vdash B : Type_*}{\Gamma \vdash \Pi(x : A), B : Type_*}\ \Pi\text{-}formation$$

$$\frac{\Gamma, x : A \vdash y : B}{\Gamma \vdash (\lambda x : A, y) : \Pi x : A, B}\ \Pi\text{-}abstraction$$

$$\frac{\Gamma \vdash f : \Pi x : A, B \qquad \Gamma \vdash y : A}{\Gamma \vdash fy : B[y/x]}\ \Pi\text{-}application$$

Lean supports defining types and functions by inductive definitions. This mechanism will be used intensively in our formalization. An inductive type is built up from a specified list of *constructors*. The syntax for defining an inductive type is as follows:

```
inductive foo : Type
| constructor_1 : ... → foo
| constructor_2 : ... → foo
...
| constructor_3 : ... → foo
```

Inductive types come with *recursors*, which are defined at the same time as the inductive types and their constructors are defined. Recursors allow users to define functions by recursion on the structure of objects whose types are inductively defined. They also provide a principle of induction as a special case where the target type is an element of `Prop`. Examples of inductively defined types and functions include:

```
-- a non-recursive inductive type
inductive bool : Type
| ff : bool
| tt : bool
-- a recursive inductive type
inductive nat
| zero : nat
| succ : nat → nat
-- an inductively defined dependent type
inductive list (T : Type u)
| nil {} : list
| cons   : T → list → list
-- boolean negation
def bnot : bool → bool
| tt := ff
| ff := tt
-- addition on natural numbers
def add : nat → nat → nat
| a  zero     := a
| a  (succ b) := succ (add a b)
-- concatenation on lists
def concat : list α → α → list α
| []     a := [a]
| (b::l) a := b :: concat l a
```

A non-recursive inductive type that contains only one constructor is called a *structure*. The constructor of a structure simply packs the list of arguments into a single piece of data. The structure command simultaneously introduces the inductive type, its constructor and recursors, as well as the projections to each of their fields. If we do not name the constructor, `mk` is used as a default. The syntax for declaring a structure is as follows:

```
structure <name> <parameters> <parent-structures> : Type :=
<constructor> :: <fields>
```

Two useful inductive types we will use extensively in our formalization are `prod` and `subtype`. `prod` is the type of ordered pairs, representing the

cartesian product of two objects. `subtype` is intended to model the subset relation in set theory. Intuitively, `subtype` (λ `x : A, P`) denotes the collection of elements of `A` that have property `P`.

```
-- abbreviated as A × B
structure prod (α : Type u) (β : Type v) :=
(fst : α) (snd : β)

/- Remark: subtype must take a Sort instead of Type because of the axiom
    strong_indefinite_description. -/
structure subtype {α : Sort u} (p : α → Prop) :=
(val : α) (property : p val)
```

Given an object α : `A × B`, the function `prod.fst` : `A × B → A` returns the data contained in the first field of α. The application of `prod.fst` can be abbreviated as α`.fst`. We can also access the data by referring to the positions of the field. For example, α.1 works in the same way α`.fst` does. Similarly, if α is an object of a subtype, then both α`.val` and α.1 returns the data contained in the `val` field of α. We will use the latter one when the names of the fields are clear from the context.

Structures can inherit fields from other structures. In other words, we can extend existing structures by adding new fields:

```
class has_mul (α : Type u) := (mul : α → α → α)
-- semigroups
structure [class] semigroup (α : Type u) extends has_mul α :=
(mul_assoc : ∀ a b c : α, a * b * c = a * (b * c))
-- commutative semigroups
structure [class] comm_semigroup (α : Type u) extends semigroup α :=
(mul_comm : ∀ a b : α, a * b = b * a)
```

We see that these structures are marked by [`class`]. This annotation tells Lean that an object of this type can be figured out implicitly by *type class inference*. If Lean sees an argument marked by square brackets like [`x : A`] in a definition and the definition of `A` is marked by [`class`], then it searches the context for an `instance` of type `A` to instantiate x. On the other hand, curly braces {`x : A`} tells lean to infer the argument from other sources such as the expected result type. This process is also known as *elaboration* [1]. Both two kinds of implicit argument can be provided explicitly by using the `@` notation.

```
#check @list.nil
```

Under the Curry-Howard isomorphism, theorem proving in Lean is nothing more than constructing objects of correct types. Logical connectives are

defined as inductive types living in the lowest type universe `Prop`.

```
inductive true : Prop
| intro : true

inductive false : Prop

inductive and (a b : Prop) : Prop
| intro : a → b → and

inductive or (a b : Prop) : Prop
| inl {} : a → or
| inr {} : b → or
```

A proof of a proposition `p : Prop` is simply an object `H : p`. To show that `p` holds, what we need to do is to concretely construct `H` from the objects in the context. For example,

```
theorem my_thm {p q : Prop} (Hp : p) (Hq : q) : p ∧ q := and.intro Hp Hq
```

If a type has only one constructor, then we can construct objects of that type using the ⟨⟩ notation, without referring to the name of its constructor. The above theorem can be proved in an alternative way as follows.

```
theorem my_thm {p q : Prop} (Hp : p) (Hq : q) : p ∧ q := ⟨Hp, Hq⟩
```

Note that from a type-theoretic perspective, a mathematical assumption `p : Prop` is no more special than any other mathematical object, say a natural number `n : ℕ`, in the sense that all these objects are data, except that `Prop` is *impredicative* and *proof-irrelevant*. A function can take a natural number `n`, as well as a mathematical assumption `H : p`. The only two properties that make `Prop` distinct are (1) impredicativity: if `B` is of type `Prop`, then so is `Πx : A, B`; and (2) proof-irrelevance: if we have H_1 : p and H_2 : p where p is of type `Prop`, then H_1 and H_2 are definitionally equal. We will make heavy use of the fact that functions can be applied to mathematical statements in our formalization. For example, see the construction of *mbs-helper* in section 3.2.2.

In order to make proofs reusable, the formalization will be structured using Lean's `section` mechanism. Theorems proved in a `section` are relativized to a set of hypothetical constants which specify the local context including variables and assumptions. As long as the hypothetical constants get instantiated properly, valid instances of the theorems proved in a section can be obtained.

```
section
```

```
parameters {p q : Prop}
parameters (Hp : p) (Hq : q)

theorem my_thm : p ∧ q := ⟨Hp, Hq⟩
end

#check @my_thm -- my_thm : ∀ {p q : Prop}, p → q → p ∧ q
```

2.2 WQO Theory

In this section, we introduce basic concepts needed to understand the proof of Kruskal's theorem. Both mathematical representations of these concepts and their encoding in Lean are given.

2.2.1 Well-quasi-orderings

Definition 2.1. Let A be a set and $\leq$ a binary relation over A. A is quasi-ordered by $\leq$ ($\leq_A$ is a quasi order) if $\leq$ is *reflexive* and *transitive*.

In our type-theoretic encoding, a quasi-ordered set is treated as a type `A : Type` equipped with an object `le : A → A → Prop` such that reflexivity `refl : ∀ a : A, le a a` and transitivity `trans : ∀ {a b c : A}, le a b → le b c → le a c` are satisfied. We can use the structure command in Lean to define quasi-ordered sets as follows:

```
structure [class] quasiorder (A : Type) extends has_le A :=
(refl : ∀ a, le a a)
(trans : ∀ {a b c}, le a b → le b c → le a c)
```

Definition 2.2. Given a quasi-order $\leq$ over a set A, an infinite sequence $(a_i)_{i\in\mathbb{N}}$ of elements of A is called *good* if there exist natural numbers i, j such that $i < j$ and $a_i \leq a_j$. A sequence is called *bad* if it is not good. A quasi-ordered set A is well quasi-ordered, abbreviated as *wqo*, if every infinite sequence of elements of A is good.

According to the definition, every well quasi-ordered set is a quasi-ordered set with an additional property saying that every infinite sequence over A is good. Just like quasi-ordered sets extend sets with a single binary relation, well quasi-ordered sets extend quasi-orderd sets. In Lean, this is represented by the following:

```
structure [class] wqo (A : Type) extends quasiorder A :=
(is_good : ∀ f : ℕ → A, ∃ i j, i < j ∧ le (f i) (f j))
```

Definition 2.3. Let $(A, \leq_A)$ and $(B, \leq_B)$ be two quasi-ordered sets. The *product order* $\leq_{prod}$ over $A \times B$ is defined such that $(a_1, b_1) \leq_{prod} (a_2, b_2)$ if and only if $a_1 \leq_A a_2$ and $b_1 \leq_B b_2$.

```
def prod_order {A B : Type} (o₁ : A → A → Prop) (o₂ : B → B → Prop)
(s : A × B) (t : A × B) := o₁ (s.1) (t.1) ∧ o₂ (s.2) (t.2)
```

It can be easily seen that if both $(A, \leq_A)$ and $(B, \leq_B)$ are quasi-ordered sets, then $A \times B$ is quasi-ordered by $\leq_{prod}$.

```
instance qo_prod  {A B: Type} [o₁ : quasiorder A] [o₂ : quasiorder B] :
  quasiorder (A × B) :=
let op : A × B → A × B → Prop  := prod_order o₁.le o₂.le in
have refl : ∀ p : A × B, op p p, by intro; apply and.intro; repeat {apply
    quasiorder.refl},
have trans : ∀ a b c, op a b → op b c → op a c, from λ x y z h1 h2,
⟨(quasiorder.trans h1.left h2.left), quasiorder.trans h1.right h2.right⟩,
show _, from quasiorder.mk (has_le.mk op) refl trans
```

Definition 2.4. Let Q be a set and $\leq_Q$ a binary relation over Q. Let A and B be subsets of Q. A mapping $f : A \to B$ is *non-descending* if $a \leq_Q f(a)$ for every $a \in A$. The class of finite subsets of Q is denoted as Q^*. Given $A, B \in Q^*$, $A \leq_* B$ if and only if there is an injective non-descending mapping from A to B.

```
def finite_subsets (Q : Type) : Type := {x : set Q // finite x}

def inj_from_to {A B: Type} (f : A → B) (S1 : set A) (S2 : set B) :=
maps_to f S1 S2 ∧ inj_on f S1

def non_descending {Q : Type} (A B : finite_subsets Q) (o : Q → Q → Prop)
(f : Q → Q) := ∀ a : Q, a ∈ A.1 → o a (f a) ∧ f a ∈ B.1

def star {Q : Type} (o : Q → Q → Prop) (A B : finite_subsets Q) :=
∃ f, inj_from_to f A.1 B.1 ∧ non_descending A B o f
```

The auxiliary definition `inj_from_to` says that f is (1) a function from S_1 to S_2, and (2) f is an injection on S_1. This definition is needed because injectivity defined on types is not expressive enough to talk about domains as sets.

2.2.2 Finite Trees

Definition 2.5. *(Finite trees)* Finite trees are sequences defined recursively as follows.

(T1) If $t_1, ..., t_n$ are finite trees, then $T = \langle t_1, ..., t_n \rangle$ is a finite tree. A pair (t_i, i) where $1 \leq i \leq n$ is called a *branch* of T. We denote the set of branches of T by $B(T)$.

(T2) Only objects obtained by (T1) are finite trees.

In our formalization, we use functions of type *fin* $n \to$ *finite-tree* to represent sequences of finite trees. Think of *fin* n as the finite set of natural numbers less than n. This encoding allows us to extract concrete information of the branches of a finite tree by applying the function to some i : *fin* n.

```
structure fin (n : nat) := (val : nat) (is_lt : val < n)

inductive finite_tree : Type
| cons : Π {n : ℕ}, (fin n → finite_tree) → finite_tree
-- to handle trees of complicated forms, e.g., (f n) where f : ℕ → finite_tree
theorem finite_tree_destruct {t : finite_tree} :
∃ n (ss : fin n → finite_tree), t = cons ss :=
finite_tree.cases_on t (λ n a, ⟨n,a,rfl⟩)

def branches_aux {n : ℕ} (ts : fin n → finite_tree) : set (finite_tree×ℕ)
:= {x : finite_tree × ℕ | ∃ a : fin n, ts a = x.1 ∧ val a = x.2}
-- returns the set of branches of a finite tree
def branches : finite_tree → set (finite_tree × ℕ)
| (@cons n ts) := branches_aux ts
```

Note that no information can be extracted from $f : fin\ 0 \to$ *finite-tree* because the type $fin\ 0$ is empty. Therefore, *cons* f behaves just like the empty sequence. Think of *cons* f as a single node. It is provable that the set of branches of a node is empty.

```
def node {ts : fin 0 → finite_tree} : finite_tree := @cons 0 ts
-- if we have an object of type fin 0, then we can prove everything
def {u} fin_zero_absurd {α : Sort u} (i : fin 0) : α :=
absurd i.2 (not_lt_zero i.1)
-- the set of branches of a node is empty
theorem empty_branches (ts : fin 0 → finite_tree) : branches_aux ts = ∅
:= have ∀ x, x ∉ branches_aux ts,
      from λ x h, let ⟨a,ha⟩ := h in fin_zero_absurd a,
  show _, from set.eq_empty_of_forall_not_mem this
```

Definition 2.6. *(Homeomorphic embedding)* A finite tree T is homeomorphically embeddable to a finite tree T', written $T \preceq_{emb} T'$, if and only if

(E1) there exists a branch R of T' such that $T \preceq_{emb} \pi_1(R)$, or

(E2) there exists an injection H such that for every branch R of T, $H(R)$ is a branch of T' and $\pi_1(R) \preceq_{emb} \pi_1 \circ H(R)$.

```
def embeds : finite_tree → finite_tree → Prop
| (@cons _ ts) (@cons _ us) := (∃ j, embeds (cons ts) (us j)) ∨
                               (∃ f, injective f ∧ ∀ i, embeds (ts i) (us (f i)))
```

Theorem 2.1. *$\preceq_{emb}$ is reflexive and transitive.*

Proof. Reflexivity: trivial by induction on finite trees. The identity function id is a witness of (E2).

Transitivity: Let T_1, T_2, T_3 be finite trees. Suppose $T_1 \preceq_{emb} T_2$ and $T_2 \preceq_{emb} T_3$. We prove by induction on T_3. If $T_2 \preceq_{emb} T_3$ via (E1), then $T_1 \preceq_{emb} T_3$ via (E1) by inductive hypothesis. Suppose that $T_2 \preceq_{emb} T_3$ via (E2). Let H be the witness. If $T_1 \preceq_{emb} T_2$ via (E1), then $T_1 \preceq_{emb} \pi_1(R)$ for some $R \in B(T_2)$. Hence $T_1 \preceq_{emb} \pi_1 \circ H(R)$ by inductive hypothesis. If $T_1 \preceq_{emb} T_2$ via (E2), then let H' be the witness. Then $H \circ H'$ witnesses that $T_1 \preceq_{emb} T_3$ via (E2) by inductive hypothesis. □

```
theorem node_embeds {ts : fin 0 → finite_tree} (t : finite_tree) :
@cons 0 ts ≼ t :=
begin
 induction t with n a ih,
 dsimp [embeds],
 pose f : fin 0 → fin n := λ i : fin 0, fin_zero_absurd i,
 apply or.inr,
 existsi f,
 split,
  intros i j hij, exact fin_zero_absurd i,
 intro i, exact fin_zero_absurd i
end
-- reflexivity of homeomorphic embedding
theorem embeds_refl (t : finite_tree) : t ≼ t :=
begin
induction t with n a ih,
cases n, apply node_embeds,
apply cons_embeds_cons_right,
apply injective_id, exact ih
end
```

```
-- transitivity of homeomorphic embedding
theorem embeds_trans_aux : ∀ {u s t}, t ≼ u → s ≼ t → s ≼ u :=
begin
 intro u,
 induction u with ul us ihu,
  intros s t, cases s with sl ss,
  cases t with tl ts,
 intro H₁, dsimp [embeds] at H₁, cases H₁ with H₁₁ H₁₂,
 cases H₁₁ with i H₁₁, intro H₂,
  apply cons_embeds_cons_left (ihu _ H₁₁ H₂),
  cases H₁₂ with f Hf, cases Hf with injf Hf,
 intro H₂, dsimp [embeds] at H₂, cases H₂ with H₂₁ H₂₂,
  cases H₂₁ with j H₂₁,
  apply cons_embeds_cons_left (ihu _ (Hf j) H₂₁),
  cases H₂₂ with g Hg, cases Hg with injg Hg,
  apply cons_embeds_cons_right,
  apply injective_comp injf injg,
  intro i, apply ihu _ (Hf (g i)) (Hg i)
end

theorem embeds_trans {s t u : finite_tree} (H₁ : s ≼ t) (H₂ : t ≼ u) : s ≼ u
:= embeds_trans_aux H₂ H₁
```

Theorem 2.2. *Let T be a finite tree. If $R \in B(T)$, then $\pi_1(R) \preceq_{emb} T$.*

Proof. Trivial by (E1) and reflexivity. However, to prove it formally in Lean, we have to construct the proof term concretely by induction on the structure of finite trees. □

```
theorem embeds_of_branches {t : finite_tree × ℕ} {T : finite_tree} : t ∈ (
    branches T) → t.1 ≼ T :=
begin
  cases T with n ts,
  intro H, cases H with a h,
  cases t.1 with t1a t1s,
  dsimp [embeds], apply or.inl,
  fapply exists.intro,
  exact a, rw h.left, apply embeds_refl
end
```

Definition 2.7. Let T be a finite tree. The *size* of T is defined to be

$$size(T) = 1 + \Sigma_{R \in B(T)} size(\pi_1(R))$$

Note that *size* is well-defined because each $\pi_1(R)$ is structurally smaller than T. It is essentially a definition by recursion on finite trees.

```
def upto (n : ℕ) : list (fin n) :=
dmap (λ i, i < n) fin.mk (list.upto n)
-- summation over lists
def Suml (f : A → ℕ) : list A → ℕ
| [] := 0
| (a :: ls) := f a + Suml ls
-- cardinality of finite trees
def size : finite_tree → ℕ
| (@cons n ts) := Suml (λ i, size (ts i)) (fin.upto n) + 1
```

The function `fin.upto` takes an `n : ℕ` and returns a list of elements of the type `fin n`. The function `Suml` takes a function f and a list l, applies f to each of the elements in l, and returns the summation.

Theorem 2.3. *If R is a branch of T, then $size(\pi_1(R)) < size(T)$.*

Proof. Trivial by definition. □

Theorem 2.3 is mathematically trivial since we know the properties of summation. However, to prove the theorem formally, we have to show that the summation `Suml` over lists does have the desired properties.

```
theorem le_of_mem_Suml {f : A → ℕ} {a : A} {l : list A} :
a ∈ l → f a ≤ Suml f l :=
begin
  induction l with b ls ih, intro h, exact absurd h (not_mem_nil _),
  dsimp [Suml], intro h, assert h' : a = b ∨ a ∈ ls, exact h,
  cases h' with l r, rw l, apply le_add_right,
  rw add_comm,apply le_add_of_le, exact ih r
end
-- an auxiliary theorem for the convenience of the proof
theorem lt_of_size_branches_aux {n : ℕ} (ts : fin n → finite_tree)
(k : fin n) : size (ts k) < Suml (λ i, size (ts i)) (upto n) + 1 :=
begin
  assert kin : k ∈ upto n, exact mem_upto n k,
  assert h : size (ts k) ≤ Suml (λ i, size (ts i)) (upto n),
  apply le_of_mem_Suml kin,
  apply lt_succ_of_le, assumption
end
-- the size of a finite tree is great than the size of any of its branches
theorem lt_of_size_of_branches {t : finite_tree × ℕ} {T : finite_tree} :
t ∈ branches T → size t.1 < size T :=
```

```
begin
  cases T with n ts,
  intro h,
  assert h' : ∃ i, ts i = t.1, cases h with b hb,
    {exact exists.intro b hb.left},
  cases h' with c hc, rw ←hc, apply lt_of_size_branches_aux
end
```

3 The Formalization

In this section, we describe the formalization of Kruskal's tree theorem. For proofs that require substantial reasoning, we will give both mathematical presentations and their formalizations in Lean. If the formal details are not interesting or tedious, we will omit their formalizations partially. For trivial statements, we will only give their formalizations. For mathematical presentations, implicit arguments, i.e., hypothetical constants in `sections`, will not be specified explicitly if they are clear from the context.

3.1 Dickson's Lemma

Dickson's lemma claims that a Cartesian product with the product order $\leq_{prod}$ defined in the last section preserves well-quasi-orderedness, i.e., if A and B are well quasi-ordered, then $A \times B$ is well quasi-ordered. In the last chapter, we have seen that the reflexivity and transitivity of $\leq_{prod}$ over $A \times B$ follows immediately from the definition of $\leq_{prod}$. Therefore, to prove Dickson's lemma it suffices to show the following.

Theorem 3.1. *If A and B are wqo, then for every sequence f of elements of $A \times B$, f is good.*

Definition 3.1. Let $(A, \leq)$ be an ordered set. We call a member $f(m)$ of a sequence f over A *terminal* if there is no $n > m$ such that $f(m) \leq f(n)$.

```
def terminal {A : Type} (o : A → A → Prop) (f : ℕ → A) (m : ℕ) :=
∀ n, m < n → ¬ o (f m) (f n)
```

Lemma 3.1. *If A is wqo, then for every sequence f over A, there are only finitely many terminal members $f(i)$ of f.*

Proof. Suppose there are infinitely many terminal members. For every $n \in \mathbb{N}$, we can find a $r > n$ such that $f(r)$ is terminal. Now we define a new sequence g as follows:

$g(0) = f(r_0)$ for some r_0 such that $r_0 \geq 0$ and $f(r_0)$ is terminal.

$g(n+1) = f(r_{n+1})$ for some r_{n+1} such that $r_{n+1} > r_n$ and $f(r_{n+1})$ is terminal.

Intuitively, g simply enumerates the terminal elements of f. We claim that g is bad. This is because if there exist i, j such that $i < j$ and $g(i) \leq g(j)$, then $f(r_i) \leq f(r_j)$ by definition. By our construction, $r_i < r_j$. This implies that $f(r_i)$ is not terminal. Contradiction. □

Let us take a close look at what we need in this proof. We first want to construct the sequence $(r_n)_{n\in\mathbb{N}}$. Note that r is essentially a function from $\mathbb{N}$ to $\mathbb{N}$. Then we have to show that r as a function is strictly increasing so that we can conclude that $r_i < r_j$ if $i < j$. Then we simply define $g(n)$ to be $f(r_n)$ and claim that for every n, $f(n)$ is terminal. Finally, we show that g is good by a proof by contradiction. Each of these steps is reflected faithfully in our formalization as follows.

To prove the theorem, we assume that A is *wqo*, and let f be a sequence over A.

```
section
parameter {A : Type}
parameter [o : wqo A]
parameter f : ℕ → A
...
end
```

Our assumption H is that there are infinitely many terminal members of f.

```
section
parameter {A : Type}
parameter [o : wqo A]
parameter f : ℕ → A

    section
    parameter H : ∀ N, ∃ r, N < r ∧ terminal o.le f r
    ...
    end
...
end
```

Now we define the sequence $(r_n)_{n\in\mathbb{N}}$. We call it `terminal_index` in the formalization. The return type of $(r_n)_{n\in\mathbb{N}}$ is a subtype of $\mathbb{N}$ such that each of its elements x satisfies the property that $x > n$ and $f(x)$ is a terminal member of f. The advantage of using subtypes is that we can refer to the properties of a return value x easily by accessing the second field of x, without reconstructing the proof outside the definition.

```
def terminal_index (n : ℕ) : {x : ℕ // n < x ∧ terminal o.le f x} :=
nat.rec_on n (let i := some (H 0) in ⟨i, (some_spec (H 0))⟩)
(λ a rec_call,
let i' := rec_call.1, i := some (H i') in
have p : i' < i ∧ terminal o.le f i, from some_spec (H i'),
have a < i', from (rec_call.2).left,
have succ a < i, from lt_of_le_of_lt this p.left,
```

```
⟨i, ⟨this, p.right⟩⟩)
```

We show formally that the function `terminal_index` is strictly increasing.

```
lemma increasing_ti {n m : ℕ} :
n < m → (terminal_index n).1 < (terminal_index m).1 :=
nat.rec_on m (λ H, absurd H dec_trivial)
(λ a ih lt,
 have disj : n < a ∨ n = a, from lt_or_eq_of_lt_succ lt,
 have (terminal_index a).1 < (terminal_index (succ a)).1, from
   (some_spec (H (terminal_index a).1)).left,
 or.elim disj (λ Hl, lt_trans (ih Hl) this) (λ Hr, by rw Hr; exact this))
```

Now we can define g and prove that there is a contradiction.

```
private def g (n : ℕ) := f (terminal_index n).1
-- every (g n) is terminal
lemma terminal_g (n : ℕ) : terminal o.le g n :=
have ∀ n', (terminal_index n).1 < n' → ¬ (f (terminal_index n).1) ≤ (f n'),
  from ((terminal_index n).2).right,
λ n' h, this (terminal_index n').1 (increasing_ti h)
-- g is a bad sequence
lemma bad_g : ¬ is_good g o.le :=
have H1 : ∀ i j, i < j → ¬ (g i) ≤ (g j), from λ i j h, (terminal_g i) j h,
suppose ∃ i j, i < j ∧  (g i) ≤ (g j),
let ⟨i,j,h⟩ := this in
have ¬ (g i) ≤ (g j), from H1 i j h.left,
show _, from this h.right
-- there is a contradiction because g is good
lemma local_contradiction : false := bad_g (wqo.is_good g)
-- we conclude that there are only finitely many terminal members of f
theorem finite_terminal : ∃ N, ∀ r, N < r → ¬ terminal o.le f r :=
have ¬ ∀ N, ∃ r, N < r ∧ @terminal A o.le f r, by apply local_contradiction,
have ∃ N, ¬ ∃ r, N < r ∧ @terminal A o.le f r, by super,
let ⟨n,h⟩ := this in
have ∀ r, n < r → ¬ @terminal A o.le f r, by super,
⟨n,this⟩
```

Using lemma 3.1, we prove Dickson's lemma as follows.

Proof. Let f be a sequence of elements of $A \times B$. Then $fst \circ f$ is a sequence of elements of A. Since A is wqo, there are only finitely many terminal members of $fst \circ f$. Then there exists an s such that for every $n > s$, $fst \circ f(s)$ is not terminal. We define a sequence h as follows.

$h(0) = a$ such that $a > s$

$h(n+1) = b$ such that $b > h(n)$ and $fst \circ f \circ h(n) \leq fst \circ f(b)$

Intuitively, h enumerates the indices of a subsequence of non-terminal members of $fst \circ f$ such that $fst \circ f \circ h(i) \leq fst \circ f \circ h(i+1)$ for every $i \in \mathbb{N}$. This is possible because we know that every $fst \circ f(n)$ is not terminal if $n > s$, which means that for every $n > s$, we can find a $b > n$ such that $fst \circ f(n) \leq fst \circ f(b)$. Note that $snd \circ f \circ h$ is an infinite sequence of elememts of B. Since B is wqo, there exist i, j such that $i < j$ and $snd \circ f \circ h(i) \leq snd \circ f \circ h(j)$. Moreover, by our construction it is clear that $h(i) < h(j)$ if $i < j$. Therefore we claim that f is good as witnessed by $h(i)$ and $h(j)$. □

Each of the steps is reflected faithfully by the following encoding in Lean. We first instantiate lemma 3.1 with $fst \circ f$ where f is the sequence over $A \times B$ in our assumption. Then we find a `sentinel` such that every element of $fst \circ f$ beyond it is not terminal.

```
section
parameters {A B : Type}
parameters [o₁ : wqo A] [o₂ : wqo B]

 section
 parameter f : ℕ → A × B
 -- apply the theorem we proved in the last section
 theorem finite_terminal_on_A :
 ∃ N, ∀ r, N < r → ¬ @terminal A o₁.le (fst ∘ f) r :=
 finite_terminal (fst ∘ f)

 def sentinel := some finite_terminal_on_A
 ...
 end
...
end
```

The function h is then defined recursively as follows. Note that the return type of the function *h-helper* is also a subtype which says that each of its elements x is greater than the sentinel and $fst \circ f(x)$ is not terminal.

```
 def h_helper (n : ℕ) :
 {x : ℕ // sentinel < x ∧ ¬ @terminal A o₁.le (fst ∘ f) x} :=
 nat.rec_on n
 (have ∃ m, sentinel < m, by apply existence_of_nat_gt,
 let i := some this in
 have ge : sentinel < i, from some_spec this,
 have ¬ @terminal A o₁.le (fst ∘ f) i,
```

```
   from (some_spec finite_terminal_on_A) i ge,
have sentinel < i ∧ ¬ terminal o₁.le (fst ∘ f) i, from ⟨ge,this⟩,
⟨i, this⟩)
(λ a rec_call, let i' := rec_call.1 in
have lt' : sentinel < i', from (rec_call.2).left,
have ¬ terminal o₁.le (fst ∘ f) i', from (rec_call.2).right,
have ∃ n, i' < n ∧ ((fst ∘ f) i') ≤ ((fst ∘ f) n),
   from lt_of_non_terminal this,
let i := some this in have i' < i, from (some_spec this).left,
have lt : sentinel < i, from lt.trans lt' this,
have ∀ r, sentinel < r → ¬ terminal o₁.le (fst ∘ f) r,
   from some_spec finite_terminal_on_A,
have ¬ terminal o₁.le (fst ∘ f) i, from this i lt,
have sentinel < i ∧ ¬ terminal o₁.le (fst ∘ f) i, from ⟨lt,this⟩,
⟨i,this⟩)

private def h (n : ℕ) : ℕ := (h_helper n).1
```

We check that h does have the properties we want, as decribed in the above proof.

```
private lemma foo (a : ℕ) :
h a < h (succ a) ∧ (fst ∘ f) (h a) ≤ (fst ∘ f) (h (succ a)) :=
have ¬ terminal o₁.le (fst ∘ f) (h a), from ((h_helper a).2).right,
have ∃ n, (h a) < n ∧ ((fst ∘ f) (h a)) ≤ ((fst ∘ f) n), from
  lt_of_non_terminal this,
show _, from some_spec this
-- h has the property we want
theorem property_of_h {i j : ℕ} : i < j → (fst ∘ f) (h i) ≤ (fst ∘ f) (h j)
:= nat.rec_on j (λ H, absurd H dec_trivial)
(λ a IH lt,
have H1 : (fst ∘ f) (h a) ≤ (fst ∘ f) (h (succ a)), from (foo a).right,
have disj : i < a ∨ i = a, from lt_or_eq_of_lt_succ lt,
or.elim disj (λ Hl, quasiorder.trans (IH Hl) H1) (λ Hr, by simp [Hr, H1]))
-- h is strictly increasing
theorem increasing_h {i j : ℕ} : i < j → h i < h j :=
nat.rec_on j
(λ H, absurd H dec_trivial)
(λ a ih lt,
have H1 : (h a) < h (succ a), from (foo a).left,
have disj : i < a ∨ i = a, from lt_or_eq_of_lt_succ lt,
or.elim disj (λ Hl, lt_trans (ih Hl) H1) (λ Hr, by simp [Hr, H1])
-- f is good
theorem good_f : is_good f (prod_order o₁.le o₂.le) :=
```

```
 have ∃ i j : ℕ, i < j ∧ (snd ∘ f ∘ h) i ≤ (snd ∘ f ∘ h) j,
    from wqo.is_good (snd ∘ f ∘ h),
 let ⟨i,j,H⟩ := this in
 have (fst ∘ f) (h i) ≤ (fst ∘ f) (h j), from property_of_h H.left,
 have Hr : (fst ∘ f) (h i) ≤ (fst ∘ f) (h j) ∧ (snd ∘ f) (h i) ≤ (snd ∘ f) (h j),
    from ⟨this, H.right⟩,
 have h i < h j, from increasing_h H.left,
 ⟨(h i), (h j), ⟨this,Hr⟩⟩
-- every f over A × B is good
theorem good_pairs (f : ℕ → A × B) : is_good f (prod_order o₁.le o₂.le) :=
good_f f
```

Finally, we gather the facts that the product order is reflexive, transitive and every sequence over ordered pairs is good to prove Dickson's lemma.

```
def wqo_prod {A B : Type} [o₁ : wqo A] [o₂ : wqo B] : wqo (A × B) :=
let op : A × B → A × B → Prop := prod_order o₁.le o₂.le in
have refl : ∀ p : A × B, op p p,
  from λ p, ⟨quasiorder.refl p.1,quasiorder.refl p.2⟩,
have trans : ∀ a b c, op a b → op b c → op a c, from λ a b c h1 h2,
 ⟨quasiorder.trans h1.left h2.left, quasiorder.trans h1.right h2.right⟩,
show _, from wqo.mk ⟨⟨op⟩,refl,trans⟩ good_pairs
```

3.2 Higman's Lemma

Higman's lemma claims that the * operator, as defined in definition 2.4, preserves well-quasi-orderedness. We first check that the * operator preserves quasi-orderedness. Let Q be a set quasi-ordered by $\leq_Q$ and $A \in Q^*$. For reflexivity, it suffices to show that there is an injective non-descending mapping $f : A \to A$. The identity function id satisfies the requirement trivially. Let $A, B, C \in Q^*$. Suppose $A \leq_* B$ and $B \leq_* C$. For transitivity, it suffices to show that there is an injective non-descending mapping $f : A \to C$. Since $A \leq_* B$ and $B \leq_* C$, there exist an injective non-descending $f : A \to B$ and an injective non-descending $g : B \to C$. We immediately see that the mapping $g \circ f : A \to C$ is injective because function composition preserves injectivity. It is also non-descending because $\leq_Q$ is transitive. Therefore, the $\leq_*$ over Q^* is a quasi-order.

```
section
parameter {Q : Type}
parameter [o : wqo Q]
-- the relation on finite_subsets Q induced by o
def sub := @star Q o.le
-- reflexivity of sub
theorem sub_refl (q : finite_subsets Q) : sub q q :=
have ∀ a : Q, a ∈ q.1 → a ≤ (id a) ∧ id a ∈ q.1,
  begin intros, split, simp, apply quasiorder.refl, simp, assumption end,
⟨id, ⟨inj_from_to_id q.1,this⟩⟩
-- transitivity of sub
theorem sub_trans (a b c : finite_subsets Q) (H1 : sub a b) (H2 : sub b c) :
sub a c := let ⟨f,hf⟩ := H1, ⟨g,hg⟩ := H2 in
have inj : inj_from_to g ∘ f a.1 c.1,
  from inj_from_to_compose hg.left hf.left,
have ∀ q : Q, q ∈ a.1 → q ≤ g ∘ f q ∧ g ∘ f q ∈ c.1, from λ q Hq,
 have le1 : q ≤ f q, from (hf.right q Hq).left,
 have fqin : f q ∈ b.1, from (hf.right q Hq).right,
 have le2 : (f q) ≤ g ∘ f q, from (hg.right (f q) fqin).left,
 have qle : q ≤ g ∘ f q, from quasiorder.trans le1 le2,
 have g ∘ f q ∈ c.1, from (hg.right (f q) fqin).right,
 ⟨qle, this⟩,
⟨g ∘ f,⟨inj,this⟩⟩
end
```

3.2.1 A Proof Sketch

Now to prove Higman's lemma, we only have to show that there is no bad sequence of elements of Q^*.

Theorem 3.2. *If Q is wqo, then there is no bad sequence of elements of Q^*.*

We start by describing a proof sketch of the theorem. The strategy is a proof by contradiction.

Proof. (sketch) Suppose there exists a bad sequence of elements of Q^*. We can construct a new bad sequence $(A_n)_{n\in\mathbb{N}}$ such that for all n, $A_0, ..., A_n$ is an initial segment of some bad sequence and for all $i \leq n$, $|A_i|$ is as small as possible. $(A_n)_{n\in\mathbb{N}}$ is called a minimal bad sequence. Since each A_i is not empty, there exists $a_i \in A_i$ for every $i \in \mathbb{N}$. Let $B_n = A_n - \{a_n\}$. We define the set

$$ClassB = \{x \mid \exists i, x = B(i)\} = ran(B)$$

It can be easily seen that the $\leq$ over $ClassB$ is reflexive and transitive because it is the same $\leq$ over Q^* by our construction. We now show that there is no bad sequence of elements of $ClassB$ by a proof of contradiction. Suppose f is a bad sequence of elements of $ClassB$. Then there exists a bad sequence $(f_{h(i)})_{i\in\mathbb{N}}$ such that $h(0) \leq h(i)$ for all i. The existence of this sequence will allow us to construct another bad sequence *comb* that contradicts the "minimality" of our minimal bad sequence. This contradiction shows that $ClassB$ is wqo.

Since $ClassB$ is wqo, we conclude that $Q \times ClassB$ is wqo by Dickson's lemma. This means that the sequence $(a_n, B_n)_{n\in\mathbb{N}}$ is good. By our construction of $(B_n)_{n\in\mathbb{N}}$, this will imply that there exist i, j such that $i < j$ and $A_i \leq A_j$ which in turn gives us a contradiction because $(A_n)_{n\in\mathbb{N}}$ is bad. □

The main effort in formalizing Higman's lemma is the construction of the minimal bad sequence. The whole argument should be formalized in a way that it is independent of the context so that it can be applied to other objects with different types and different notions of cardinality. Moreover, the formalization of the argument should be broken into pieces so that each step of the argument become reusable and maintainable. In the next section, we decribe such a formalization in Lean.

Before diving into the details of the minimal bad sequence argument, we first present some mechanisms needed for the later formalization. A few reflections on the above proof sketch tells us that we need a measure for calculating the size of our elements and a function f that picks out a least element from a set given that measure. In the case of Higman's lemma,

the measure is the cardinality on (finite) sets and the f is the function that returns the least natural number of a set of natural numbers according to the least number principle. We give a formalization of the least number principle in Lean as follows.

```
lemma wf_aux {A : set ℕ} (n : ℕ) : n ∈ A → ∃ a, a ∈ A ∧ ∀ b,  b ∈  A → a ≤ b
:= @complete_induction_on n (λ x, x ∈ A → ∃ a, a ∈  A ∧ ∀ b, b ∈ A → a ≤ b)
(λ k ih h, by_cases
(suppose ∃ m, m ∈ A ∧ m <k, let ⟨m, Hmem, Hlt⟩ := this in ih m Hlt Hmem)
(λ Hn, have ∀ m, m ∈  A →  ¬ m < k, by super,
  ⟨k, h, (λ m h, le_of_not_gt (this m h))⟩))
-- the least number principle
theorem wf_of_le (S : set ℕ) (H : S ≠ ∅) : ∃ a, a ∈ S ∧ ∀ b, b ∈ S → a ≤ b :=
let ⟨n, Hn⟩ := exists_mem_of_ne_empty H in wf_aux n Hn
```

The least number principle `wf_of_le` induces a function `least` which takes a set S of natural numbers and an assumption that $S \neq \emptyset$, and returns the least element in S. Note that `least` is not constructive, as marked by `noncomputable`, because it requires the choice axiom to "compute" the return value.

```
noncomputable def least (S : set ℕ) (H : S ≠ ∅) : ℕ :=
some (wf_of_le S H)
```

Having formalized the least number principle, we prove the following theorem saying that given a nonempty set of functions `S : set (ℕ → ℕ)` and a number `n : ℕ`, there exists a function $f \in S$ such that $f(n) \leq g(n)$ for all $g \in S$.

```
theorem least_seq_at_n {S : set (ℕ → ℕ)} (H : S ≠ ∅) (n : ℕ) :
∃ f, f ∈ S ∧ ∀ g, g ∈ S → f n ≤ g n :=
let T : set ℕ := {x | ∃ f, f ∈ S ∧ f n = x} in
have ∃ f, f ∈ S, from exists_mem_of_ne_empty H,
let ⟨f,h⟩ := this in
have nemp : T ≠ ∅, from set.ne_empty_of_mem ⟨f,⟨h,rfl⟩⟩,
let a := least T nemp in
have a ∈ T, from least_is_mem T nemp,
let ⟨f',h⟩ := this in
have ∀ g, g ∈ S → f' n ≤ g n, from λ g Hg,
 have a ≤ g n, from minimality _ _ ⟨g,⟨Hg,rfl⟩⟩,
 by super,
⟨f',⟨h.left, this⟩⟩
```

To construct the minimal bad sequence so that any of its initial segments is an initial segment of some bad sequence, we want to be able to talk about extensions of sequences. Given two sequences $f : \mathbb{N} \to A$ and $g : \mathbb{N} \to A$, we

say that f extends g at n if for every $m \leq n$, g and f agree on the values at m.

```
def extends_at {A : Type} (n : ℕ) (f : ℕ → A) (g : ℕ → A) : Prop :=
∀ m ≤ n, g m = f m
```

We can show that `extends_at` is reflexive and transitive:

```
theorem extends_at.refl {A : Type} {n : ℕ} {f : ℕ → A} : extends_at n f f
:= λ m H, rfl
-- note that this holds when n ≤ m
theorem extends_at.trans {A : Type} {n m : ℕ} {f g h: ℕ → A}
(H1 : extends_at n f g) (H2 : extends_at m g h) (H3 : n ≤ m) :
extends_at n f h :=
λ k H, have g k = f k, from H1 k H,
have k ≤ m, from nat.le_trans H H3,
have h k = g k, from H2 k this,
by super
```

3.2.2 The Minimal Bad Sequence Argument

In this section, a general construction of the minimal bad sequence argument is given. Lean's `section` mechanism allows us to define general hypothesis and prove facts under these assumptions. To apply the results proved in these sections, one only needs to provide concrete instances of the hypothetical constants declared at the beginning of the `section`. This mechanism gives us a convenient way of managing large and complicated proofs so that they become maintainable and the intermediate steps along the way become reusable.

We first define a function *min-func* which, given an $n \in \mathbb{N}$ and an assumption that there exists a function $f : \mathbb{N} \to A$ satisfying certain property P, returns a function $f : \mathbb{N} \to A$ such that f satisfies P and $|f(n)| \leq |g(n)|$ for every g satisfying P, under some measure $|\cdot| : A \to \mathbb{N}$ which computes the size of an object $a : A$.

```
section
parameter {A : Type}
parameter {P : (ℕ → A) → Prop}
parameter g : A → ℕ
parameter H : ∃ f : ℕ → A, P f
-- the collection of all the functions that satisfy P
def colle : set (ℕ → A) := {f | P f}
-- we know that the collection is not empty by H
lemma nonempty_colle : colle ≠ ∅ :=
```

```
let ⟨a,h⟩ := H in set.ne_empty_of_mem h
-- convert each function in colle to a function from ℕ to ℕ
private def S : set (ℕ → ℕ) := image (λ f, g ∘ f) colle
-- similarly, S is not empty
lemma nonempty_S : S ≠ ∅ := image_nonempty nonempty_colle
-- apply the previous theorem
theorem exists_min_func (n : ℕ) : ∃ f, f ∈ S ∧ ∀ g, g ∈ S → f n ≤ g n :=
least_seq_at_n nonempty_S n
-- return the witness of the above theorem in its original form
def min_func (n : ℕ) : ℕ → A :=
let fc := some (exists_min_func n) in
have fc ∈ S ∧ ∀ g, g ∈ S → fc n ≤ g n,
  from (some_spec (exists_min_func n)),
some this.left
end
```

The following two theorems are trivial and we give their formalizations instead of mathematical proofs.

Theorem 3.3. *For every n, P holds for min-$func(n)$.*

```
theorem min_func_property (n : ℕ) : P (min_func n) :=
let fc := some (exists_min_func n) in
let ⟨l,r⟩ := some_spec (exists_min_func n) in
have min_func n ∈ colle ∧ (λ f, g ∘ f) (min_func n) = fc, from some_spec l ,
this.left
```

Theorem 3.4. *Let $f : \mathbb{N} \to A$ be a function satisfying P. For every n, $|min\text{-}func(n)(n)| \leq |f(n)|$.*

```
theorem min_func_minimality (f : ℕ → A) (Hp : P f) (n : ℕ) :
g (min_func n n) ≤ g (f n) :=
let fc := some (exists_min_func n) in
let ⟨l,r⟩ := some_spec (exists_min_func n) in
have min_func n ∈ colle ∧ (λ f, g ∘ f) (min_func n) = fc, from some_spec l,
have (λ f, g ∘ f) (min_func n) = fc, from this.right,
have eq2 : (λ f, g ∘ f) (min_func n) n = fc n, by rw this,
have Hr : ∀ g, g ∈ S → fc n ≤ g n,
  from (some_spec (exists_min_func n)).right,
have le : fc n ≤ (λ f, g ∘ f) f n, from Hr _ ⟨f,⟨Hp,rfl⟩⟩,
have (λ f, g ∘ f) (min_func n) n ≤ (λ f, g ∘ f) f n, by rw -eq2 at le;exact le,
by super
```

Note that if we go outside the section, the function *min-func* will take three explicit arguments. It needs a measure $|\cdot| : A \to \mathbb{N}$, an assumption $H : \exists f, P\ f$ where P is an implicit predicate of type $(\mathbb{N} \to A) \to Prop$ and a natural number $n : \mathbb{N}$ to compute the return value.

Next we construct a sequence *mbs-helper* of functions satisfying some property P such that the $(n+1)$th function in the sequence extends its predecessor at n and is the minimal one at $n+1$ in the sense of Theorem 3.4. As above, we assume the existence of a measure $|\cdot| : A \to \mathbb{N}$ and an assumption $H : \exists f, P\ f$. Then *mbs-helper* is defined recursively as follows.

$$mbs\text{-}helper(0) = min\text{-}func(|\cdot|, H, 0)$$

Suppose that we have defined $mbs\text{-}helper(n)$. We first claim there exists a function f that extends *mbs-helper* at n, and satisfies P. This is clear because $mbs\text{-}helper(n)$ is a witness by the reflexivity of `extends_at` and Theorem 3.3. Therefore, we have

$$H' : \exists f, extends\text{-}at\ n\ (mbs\text{-}helper(n))\ f \wedge P\ f$$

Now we apply *min-func* to H' to get $mbs\text{-}helper(n+1)$.

$$mbs\text{-}helper(n+1) = min\text{-}func(|\cdot|, H', n+1)$$

```
noncomputable def mbs_helper (n : ℕ) : {f : ℕ → A // P f} :=
nat.rec_on n
(let f₀ := min_func g H 0 in
 have P f₀, from min_func_property g H 0,
 ⟨f₀,this⟩)
(λ pred rec_call,
let f' := rec_call.1 in
have H1 : extends_at pred f' f', from extends_at.refl,
have H2 : P f', from rec_call.2
have HP : ∃ f, extends_at pred f' f ∧ P f, from ⟨f',⟨H1,H2⟩⟩,
let fn := min_func g HP (succ pred) in
have extends_at pred f' fn ∧ P fn, from min_func_property g HP (succ pred),
have P fn, from this.right,
⟨fn,this⟩)
```

By our construction, $mbs\text{-}helper(n+1)$ always extends $mbs\text{-}helper(n)$ at n for every n. Then it is not hard to see by a straightforward induction that for every n, m, if $m \leq n$ then $mbs\text{-}helper(n)$ extends $mbs\text{-}helper(m)$ at m.

Theorem 3.5. *For every n, P holds for $mbs\text{-}helper(n)$.*

Proof. By our construction, this follows immediately from Theorem 3.3. □

```
section
parameter n : ℕ
-- some abbreviations and facts about mbs_helper
def helper_elt := (mbs_helper n).1
def helper_succ := (mbs_helper (succ n)).1
lemma helper_ext_refl : extends_at n helper_elt helper_elt :=
extends_at.refl
lemma helper_has_property : P helper_elt := (mbs_helper n).2
lemma helper_inner_hyp : ∃ g, extends_at n helper_elt g ∧ P g :=
⟨helper_elt, ⟨helper_ext_refl, helper_has_property⟩⟩
theorem succ_ext_of_mbs_helper : extends_at n helper_elt helper_succ
:= (min_func_property g helper_inner_hyp (succ n)).left
end
```

Now we move on to the definition of the minimal bad sequence. In this `section`, our hypothetical constants are a measure $|\cdot| : A \to \mathbb{N}$, an implicit ordering $o : A \to A \to Prop$ and an assumption $H : \exists f, \neg\ is\text{-}good\ f\ o$ saying that there is a bad sequence.

```
section
-- construction and properties of mbs.
parameter {A : Type}
parameter {o : A → A → Prop}
parameter g : A → ℕ
parameter H : ∃ f : ℕ → A, ¬ is_good f o
...
end
```

We first obtain a sequence *seq-of-bad-seq* of bad sequences by instantiating the general H in the definition of *mbs-helper* with the concrete one in this `section`. Note that the predicate P in the general H is now instantiated implicitly to be $(\lambda x, \neg\ is\text{-}good\ x\ o)$.

$$seq\text{-}of\text{-}bad\text{-}seq(n) = mbs\text{-}helper(|\cdot|, H, n)$$

```
noncomputable def seq_of_bad_seq (n : ℕ) : {f : ℕ → A // ¬ is_good f o} :=
mbs_helper g H n
```

Theorem 3.6. *For every n, seq-of-bad-seq(n) is bad.*

Proof. Since P is instantiated with $(\lambda x, \neg\ is\text{-}good\ x\ o)$, by our construction this follows trivially from Theorem 3.5 □

Intuitively, *seq-of-bad-seq* is of the following form:

$$\begin{aligned}
\textit{seq-of-bad-seq}(0) &= s_{00} \quad s_{01} \quad s_{02} \quad s_{03} \cdots \\
\textit{seq-of-bad-seq}(1) &= s_{00} \quad s_{11} \quad s_{12} \quad s_{13} \cdots \\
\textit{seq-of-bad-seq}(2) &= s_{00} \quad s_{11} \quad s_{22} \quad s_{23} \cdots \\
\textit{seq-of-bad-seq}(3) &= s_{00} \quad s_{11} \quad s_{22} \quad s_{33} \cdots \\
&\cdots
\end{aligned}$$

The minimal bad sequence, *minimal-bad-seq*, is then defined to be the diagonal sequence of *seq-of-bad-seq*.

$$\textit{minimal-bad-seq}(n) = \textit{seq-of-bad-seq}(n)(n)$$

```
def minimal_bad_seq (n : ℕ) : A :=  (seq_of_bad_seq n).1 n
```

Theorem 3.7. *minimal-bad-seq is bad.*

Proof. Suppose that it is good. Then there exist i, j such that $i < j$ and

$$\textit{minimal-bad-seq}(i) \leq \textit{minimal-bad-seq}(j)$$

Since $i < j$, *seq-of-bad-seq*(j) and *seq-of-bad-seq*(i) agree on the value at i because *seq-of-bad-seq*(j) extends *seq-of-bad-seq*(i) at i. Then we know that

$$\textit{seq-of-bad-seq}(i)(i) = \textit{seq-of-bad-seq}(j)(i) \leq \textit{seq-of-bad-seq}(j)(j)$$

But this just says that *seq-of-bad-seq*(j) is good, contradicting Theorem 3.6. □

```
theorem badness_of_mbs : ¬ is_good minimal_bad_seq o :=
suppose is_good minimal_bad_seq o,
let ⟨i,j,h⟩ := this in
have i ≤ j, from le_of_lt_or_eq (or.inl h.left),
have ext : extends_at i (seq_of_bad_seq i).1 (seq_of_bad_seq j).1,
  from ext_of_seq_of_bad_seq j i this,
have i ≤ i, from nat.le_refl i,
have (seq_of_bad_seq j).1 i = (minimal_bad_seq i), from ext i this,
have o ((seq_of_bad_seq j).1 i) (minimal_bad_seq j),
  by rw this; exact h.right,
have i < j ∧ o ((seq_of_bad_seq j).1 i) ((seq_of_bad_seq j).1 j),
  from ⟨h.left, this⟩,
have good : is_good (seq_of_bad_seq j).1 o, from ⟨i,⟨j, this⟩⟩,
have ¬ is_good (seq_of_bad_seq j).1 o, from (seq_of_bad_seq j).2,
this good
```

Theorem 3.8. *If f is a bad sequence, then $|\textit{minimal-bad-seq}(0)| \leq |f(0)|$.*

Proof. Follows immediately from Theorem 3.4. □

```
theorem minimality_of_mbs_0 (f : ℕ → A) (Hf : ¬ is_good f o) :
g (minimal_bad_seq 0) ≤ g (f 0) := min_func_minimality g H f Hf 0
```

Theorem 3.9. *If f is a bad sequence and f extends minimal-bad-seq at n, then $|\textit{minimal-bad-seq}(n+1)| \leq |f(n+1)|$.*

Proof. Since f extends *minimal-bad-seq* at n, it extends *seq-of-bad-seq*(n) at n by definition. Define $\varphi(f) :=$ f is bad and f extends *seq-of-bad-seq*(n) at n. Instantiating the P in Theorem 3.4 with the φ here gives us the result. Note that the conclusion might not be true if f does not extend *minimal-bad-seq* at n. □

```
theorem minimality_of_mbs (n : ℕ) (f : ℕ → A)
 (H1 : extends_at n minimal_bad_seq f ∧ ¬ is_good f o) :
 g (minimal_bad_seq (succ n)) ≤ g (f (succ n)) :=
have Hl : ∀ m, m ≤ n →  f m = (bad_seq_elt n) m, from λ m Hle,
  have f m = minimal_bad_seq m, from H1.left m Hle,
  have bad_seq_elt n m = minimal_bad_seq m,
    from congruence_of_seq_of_bad_seq Hle,
  by super,
have ins_P : extends_at n (bad_seq_elt n) f ∧ ¬ is_good f o,
  from ⟨Hl, H1.right⟩,
have g (min_func g (bad_seq_inner_hyp n) (succ n) (succ n)) ≤ g (f (succ n))
   ,
  from min_func_minimality g (bad_seq_inner_hyp n) f ins_P (succ n),
by super
```

Now we have completed the construction of the minimal bad sequence. Recall that we only need two explicit hypothetical constants to get the minimal bad sequence. One is the measure $|\cdot| : A \to \mathbb{N}$, the other is an assumption that there exists a bad sequence

$$H : \exists f, \neg \textit{is-good } f \ o$$

Implicitly, we are also assuming that there is an underlying type A and an ordering $o : A \to A \to \textit{Prop}$ over it. But we do not have to specify them explicitly because Lean can infer their existence from the context.

To follow the proof sketch given in the last section, we have to construct the bad sequence *comb* in order to get a contradiction. We begin by developing a general mechanism for concatenating bad sequences so that a new

bad sequence can be obtained. In this `section`, we assume that there exist two bad sequences f and g, and there exists a function $h : \mathbb{N} \to \mathbb{N}$ such that the following two conditions are satisfied:

$$Hh : \forall i, h(0) \leq h(i)$$

$$H : \forall i\ j, f(i) \leq (g(j - h(0))) \to f(i) \leq (f(h(j - h(0))))$$

The second condition seems to be a bit subtle at first glance. The intuition behind H will be made clear as we move on. But for now let us see how a new bad sequence can be constructed with these hypothetical constants. We define

$$comb(n) = \begin{cases} f(n) & if\ \ h(0) \neq 0 \wedge n < h(0), \\ g(n - h(0)) & otherwise; \end{cases}$$

```
section
/-- Given two bad sequences f and g, and a function h which modifies indices,
    construct a new sequence by concatenating f and g at (h 0). --/
parameter {Q :Type}
parameter {o : Q → Q → Prop}
parameters f g : ℕ → Q
parameter h : ℕ → ℕ
parameter Hh : ∀ i, h 0 ≤ h i
parameter Hf : ¬ is_good f o
parameter Hg : ¬ is_good g o
parameter H : ∀ i j, o (f i) (g (j - h 0)) → o (f i) (f (h (j - h 0)))

def comb (n : ℕ) : Q := if h 0 ≠ 0 ∧ n ≤ pred (h 0) then f n else g (n - (h 0))
...
end
```

Theorem 3.10. *comb is bad.*

Proof. Suppose that it is good. Then there exist i, j such that $i < j$ and

$$comb(i) \leq comb(j)$$

We prove by cases on the values of $h(0)$.

Suppose $h(0) = 0$, then $comb = g$ by definition. Since g is bad, $comb$ is bad.

Suppose $i < h(0)$ and $j < h(0)$. Then we have

$$f(i) = comb(i) \leq comb(j) = f(j)$$

by definition. But this just says that f is good, contradicting our assumption.

Suppose $i < h(0)$ and $j \geq h(0)$. Then we have

$$f(i) = comb(i) \leq comb(j) = g(j - h(0))$$

By H, we have

$$f(i) \leq f(h(j - h(0)))$$

By Hh, we have

$$h(0) \leq h(j - h(0))$$

Since $i < h(0)$, we have

$$i < h(j - h(0))$$

But this just says that f is good as witnessed by i and $h(j - h(0))$. Contradiction.

Suppose $i \geq h(0)$ and $j < h(0)$. Then $j < i$, which contradicts our assumption that $i < j$.

Suppose $i \geq h(0)$ and $j \geq h(0)$. Then we have

$$g(i - h(0)) = comb(i) \leq comb(j) = g(j - h(0))$$

Since $i < j$, we have

$$i - h(0) < j - h(0)$$

But this just says that g is good as witnessed by $i - h(0)$ and $j - h(0)$. Contradiction.

All the cases lead to a contradiction. Therefore, *comb* must be bad. □

```
theorem bad_comb : ¬ is_good comb 0 :=
λ good, let ⟨i,j,hw⟩ := good in
by_cases (...) (...)
```

Intuitively, *comb* is obtained by concatenating f and g at index $h(0)$, i.e, $comb(h(0)) = g(0)$. In the later proofs, the sequence g here will be a subsequence (indexed by the function h) of some "larger" sequence G. Think of $g(n) = G \circ h(n)$. The hypothesis H simply says that if $f(i) \leq g(k)$ then $f(i) \leq f(h(k))$. In the later proofs we will construct the sequence G so that $G(m) \leq f(m)$ for every m. This implies that H holds because if $f(i) \leq g(j - h(0))$, then

$$f(i) \leq g(j - h(0)) = G \circ h(j - h(0)) \leq f(h(j - h(0)))$$

Having the minimal bad sequence and the combined sequence *comb* in hand, we can obtain a contradiction by adding a new assumption Hbp : $|g(0)| < |minimal\text{-}bad\text{-}seq(h(0))|$. Note that we still need the assumption

Hex : $\exists f, \neg$ *is-good* f o to construct the minimal bad sequence. We also instantiate the f in the above `section` with *minimal-bad-seq*. Now define

$$\textit{comb-seq-with-mbs} = \textit{comb}(\textit{minimal-bad-seq}, g, h)$$

```
section
parameter {Q :Type}
parameter {o : Q → Q → Prop}
parameters {g : ℕ → Q}
parameter h : ℕ → ℕ
parameter m : Q → ℕ -- the measure
parameter Hh : ∀ i, h 0 ≤ h i
parameter Hex : ∃ f, ¬ is_good f o
parameter Hg : ¬ is_good g o
parameter H : ∀ i j, o (minimal_bad_seq m Hex i) (g (j − h 0)) →
  o (minimal_bad_seq m Hex i) ((minimal_bad_seq m Hex) (h (j − h 0)))
parameter Hbp : m (g 0) < m (minimal_bad_seq m Hex (h 0))

def comb_seq_with_mbs := comb (minimal_bad_seq m Hex) g h
...
end
```

Theorem 3.11. *comb-seq-with-mbs*$(h(0)) = g(0)$

Proof. Since $h(0) \not< h(0)$, we have *comb-seq-with-mbs*$(h(0)) = g(0)$ by definition of *comb*. □

```
lemma comb_seq_h0 : comb_seq_with_mbs (h 0) = g 0 := ...
```

Theorem 3.12. *comb-seq-with-mbs is bad.*

Proof. Immediately follows from Theorem 3.10. □

```
theorem bad_comb_seq_with_mbs : ¬ is_good comb_seq_with_mbs o :=
bad_comb (minimal_bad_seq m Hex) g h Hh (badness_of_mbs m Hex) Hg H
```

Theorem 3.13. *If* $h(0) \neq 0$, *comb-seq-with-mbs extends minimal-bad-seq at* $h(0) - 1$.

Proof. It suffices to show that for all $n \leq h(0) - 1$, *comb-seq-with-mbs*$(n) =$ *minimal-bad-seq*(n). Since $n \leq h(0) - 1$, we have $n < h(0)$. By definition, *comb-seq-with-mbs*$(n) =$ *minimal-bad-seq*(n). □

```
theorem comb_seq_extends_mbs_at_pred_bp (H : h 0 ≠ 0):
extends_at (pred (h 0)) (minimal_bad_seq m Hex) comb_seq_with_mbs :=
λ m Hm, if_pos ⟨H, Hm⟩
```

Theorem 3.14. *In the context of this* `section`*, there is a contradiction.*

Proof. We prove by cases on the values of $h(0)$. Suppose $h(0) = 0$. By Theorem 3.11 we have

$$\textit{comb-seq-with-mbs}(0) = \textit{comb-seq-with-mbs}(h(0)) = g(0)$$

By *Hbp*, we have

$$|\textit{comb-seq-with-mbs}(0)| < |\textit{minimal-bad-seq}(0)|$$

Since *comb-seq-with-mbs* is a bad sequence, this contradicts Theorem 3.8.

Suppose $h(0) \neq 0$. By Theorem 3.9 and Theorem 3.13, we have

$$|\textit{minimal-bad-seq}(h(0))| \leq |\textit{comb-seq-with-mbs}(h(0))|$$

By Theorem 3.11, we have

$$|\textit{minimal-bad-seq}(h(0))| \leq |g(0)|$$

But this contradicts *Hbp*. □

```
theorem local_contra_of_comb_seq_with_mbs : false :=
by_cases
(assume eq0 : h 0 = 0, ...)
(assume Hneg, ...)
```

Let us take a close look at what we have done. We have proved that given the assumption that there exists a bad sequence, if there exists a bad sequence g and a function h such that Hh,H and Hbp are satisfied, then there is a contradiction. In other words, we have developed a general mechanism for obtaining a contradiction. In the later proofs, we will get contradictions simply by constructing concrete instances of g and h and showing that they satisfy Hh, H and Hbp, which are trivial facts by our construction. This keeps the formalization concise because there is no need to write down the proof again if the strategy is similar.

We further develop a mechanism for obtaining the h satisfying Hh, assuming that there exist a sequence G and a function f such that $G \circ f$ is a bad sequence.

Theorem 3.15. *There exists a function h such that $G \circ h$ is a bad sequence and $\forall i, h(0) \leq h(i)$.*

Proof. By the least number principle, we can take a least element $s \in ran(f)$. Since $\exists m, s = f(m)$, we take such an m. It is clear that $f(m) \leq f(n)$ for every n. Define $h(n) = f(m+n)$. $G \circ h$ is bad because otherwise there exist i, j such that

$$G \circ f(m+i) \leq G \circ f(m+j)$$

which contradicts the assumption that $G \circ f$ is bad. □

```
theorem exists_sub_bad :
∃ h : ℕ → ℕ, ¬ is_good (f ∘ h) o ∧ ∀ i : ℕ, h 0 ≤ h i :=
have badness : ¬ is_good (f ∘ h) o, from
  suppose is_good (f ∘ h) o,
  let ⟨i,j,hij⟩ := this in
  have index_of_min + i < index_of_min + j,
      from add_lt_add_left (and.left hij) _,
  have is_good (f ∘ g) o,
      from ⟨index_of_min + i,⟨index_of_min + j,⟨this,hij.right⟩⟩⟩,
  H this,
have ∀ i : ℕ, h 0 ≤ h i, from λ i, minimality_of_min (index_of_min + i),
⟨h,⟨badness,this⟩⟩
```

3.2.3 Higman's Lemma Continued

We now decribe the formalization of Higman's lemma following the sketch given in the beginning of section 3. In this `section`, we fix the measure $|\cdot|$ to be the cardinality on finite sets and assume that there exists a bad sequence of elements of Q^*.

```
section
parameter {Q : Type}
parameter [o : wqo Q]
parameter H : ∃ f : ℕ → finite_subsets Q, ¬ is_good f sub
def card_of_finite_subsets {A : Type} (s : finite_subsets A) := card s.1
...
end
```

Since there exists a bad sequence, we can define the minimal bad sequence of elements of Q^*.

$$Higman\text{-}mbs = minimal\text{-}bad\text{-}seq$$

It follows immediately from Theorem 3.7 that *Higman-mbs* is bad.

Theorem 3.16. *For every n, $Higman\text{-}mbs(n) \neq \emptyset$.*

Proof. Suppose $Higman\text{-}mbs(i) = \emptyset$ for some i. We prove that

$$Higman\text{-}mbs(i) \leq Higman\text{-}mbs(i+1)$$

by showing that there exists an injective and non-descending function from $Higman\text{-}mbs(i)$ to $Higman\text{-}mbs(i+1)$. It is clear that the identity function id is injective. id is non-descending because there is no element in $Higman\text{-}mbs(i)$. Since $i < i+1$, i and $i+1$ witness that $Higman\text{-}mbs$ is good, but this contradicts the fact that $Higman\text{-}mbs$ is bad. □

```
theorem nonempty_mem_of_mbs (n : ℕ) : (Higman's_mbs n).1 ≠ ∅ :=
suppose (Higman's_mbs n).1 = ∅,
have is_good Higman's_mbs sub, from ...,
badness_of_Higman's_mbs this
```

Since each element of $Higman\text{-}mbs$ is not empty, we can select an a_n from each $Higman\text{-}mbs(n)$. Define $B(n) = Higman\text{-}mbs(n) - \{a_n\}$.

```
def B_pairs (n : ℕ) : Q × finite_subsets Q :=
have ∃ a : Q, a ∈ (Higman's_mbs n).1,
  from exists_mem_of_ne_empty (nonempty_mem_of_mbs n),
let q := some this in
let b := (Higman's_mbs n).1 \ insert q ∅ in
have finite (Higman's_mbs n).1, from (Higman's_mbs n).2,
have finite b, from @finite_diff _ _ _ this,
(q, ⟨b,this⟩)

private def B (n : ℕ) : finite_subsets Q := (B_pairs n).2
```

Theorem 3.17. *For every i, j, if $Higman\text{-}mbs(i) \leq B(j)$ then*

$$Higman\text{-}mbs(i) \leq Higman\text{-}mbs(j)$$

Proof. Since $Higman\text{-}mbs(i) \leq B(j)$, there exists an injective and non-descending f from $Higman\text{-}mbs(i)$ to $B(j)$. Then $id \circ f$ is an injective and non-descending function from $Higman\text{-}mbs(i)$ to $Higman\text{-}mbs(j)$. This is because id is an injective and non-descending function from $B(j)$ to $Higman\text{-}mbs(j)$. □

```
theorem trans_of_B (i j : ℕ) (H1 : sub (Higman's_mbs i) (B j)) :
sub (Higman's_mbs i) (Higman's_mbs j) := ...
```

Theorem 3.18. *If there exists a function h such that $B \circ h$ is bad and $\forall i, h(0) \leq h(i)$, then there is a contradiction.*

Proof. By Theorem 3.14, it suffices to construct a bad g and a function h such that Hh, H and Hbp are satisfied. Let h be the one in our assumption and $g = B \circ h$. Hh holds by assumption. $B \circ h$ is bad by assumption. To show that H holds, let i, j be arbitrary and assume that $Higman\text{-}mbs(i) \leq B \circ h(j - h(0))$. By Theorem 3.17, we have

$$Higman\text{-}mbs(i) \leq Higman\text{-}mbs(h(j - h(0)))$$

To show that Hbp holds, it suffices to show that

$$|B \circ h(0)| < |Higman\text{-}mbs(h(0))|$$

This is trivial because $|B(n)| < |Higman\text{-}mbs(n)|$ for every n. □

```
section
parameter Hg : ∃ g : ℕ → ℕ, ¬ is_good (B ∘ g) sub ∧ ∀ i : ℕ, g 0 ≤ g i
private def g := some Hg

theorem Higman's_Hg : ¬ is_good (B ∘ g) sub :=
let ⟨l,r⟩ := some_spec Hg in l

theorem Higman's_Hex : ∃ f, ¬ is_good f sub := ⟨(B ∘ g),Higman's_Hg⟩

theorem Higman's_Hh : ∀ i : ℕ, g 0 ≤ g i := (some_spec Hg).right

theorem Higman's_H : ∀ i j, sub (Higman's_mbs i) ((B ∘ g) (j - g 0)) →
sub (Higman's_mbs i) (Higman's_mbs (g (j - g 0))) :=
λ i j, λ H1, trans_of_B i (g (j - g 0)) H1
-- auxiliary theorem for proving Higman's_ Hbp
theorem card_B_lt_mbs (n : ℕ) : card (B n).val < card (Higman's_mbs n).val
:= ...
theorem Higman's_Hbp : card_of_finite_subsets (B (g 0)) <
  card_of_finite_subsets (Higman's_mbs (g 0)) :=
card_B_lt_mbs (g 0)
-- a one-line proof
theorem Higman's_local_contradition : false :=
local_contra_of_comb_seq_with_mbs ...
```

Now we define $ClassB = ran(B)$. It is defined to be a `Type` because we want to show that $ClassB$ is wqo. Recall that a wqo is a `structure`.

```
def ClassB : Type := {x : finite_subsets Q // ∃ i, B i = x}
```

Theorem 3.19. *There is no bad sequence of elements of ClassB.*

Proof. Suppose there exists a bad sequence. This sequence must be a sequence of the form $B \circ f$ for some f. By Theorem 3.15, there exists a function h such that $B \circ h$ is bad and $\forall i, h(0) \leq h(i)$. By Theorem 3.18, we have a contradiction. □

```
def oB (b1 : ClassB) (b2 : ClassB) : Prop := sub b1.val b2.val

  section
  parameter HfB : ∃ f, ¬ is_good f oB
  ...
  theorem exists_sub_bad_B_seq :
  ∃ h : ℕ → ℕ, ¬ is_good (B ∘ h) sub ∧ ∀ i : ℕ, h 0 ≤ h i :=
  exists_sub_bad ...
  end
--every sequence over ClassB is good
theorem oB_is_good : ∀ f, is_good f oB :=
by_contradiction
(suppose ¬ ∀ f, is_good f oB,
have ∃ f, ¬ is_good f oB, from classical.exists_not_of_not_forall this,
have ∃ h : ℕ → ℕ, ¬ is_good (B ∘ h) sub ∧ ∀ i : ℕ, h 0 ≤ h i,
  from exists_sub_bad_B_seq this,
Higman's_local_contradition this)
```

Since the $\leq$ over *ClassB* is the same one as the $\leq$ over Q^*, we have reflexivity and transitivity for free. Therefore, *ClassB* is wqo. By Dickson's lemma, $Q \times ClassB$ is wqo.

Theorem 3.20. *Higman-mbs is good.*

Proof. Consider the sequence $(a_n, B_n)_{n\in\mathbb{N}}$. Since $Q \times ClassB$ is wqo, there exist i, j such that $i < j$ and $a_i \leq a_j$ and $B_i \leq B_j$. Then there exists an injective and non-descending f_1 from $B(i)$ to $B(j)$. We define

$$f_2(a) = \begin{cases} a_j & if \quad a = a_i, \\ f_1(a) & otherwise; \end{cases}$$

to be a function from *Higman-mbs*(i) to *Higman-mbs*(j). It can be seen easily that f_2 is non-descending because $a_i \leq a_j$ and f_1 is non-descending. Now we show that f_2 is an injection:

Let $x_1, x_2 \in Higman\text{-}mbs(i)$. Suppose $x_1 \neq x_2$.

Suppose $x_1 = a_i$. Then $f_2(x_2) = f_1(x_2) \in B(j)$. Since $f_2(x_1) = a_j \notin B(j)$, $f(x_1) \neq f(x_2)$.

Suppose $x_1 \neq a_i$. Suppose $x_2 = a_i$. then the situation is similar to the above case. Suppose $x_2 \neq a_i$. Then $f_2(x_1) = f_1(x_1)$ and $f_2(x_2) = f_1(x_2)$. Since f_1 is injective, $f_2(x_1) \neq f_2(x_2)$. □

```
theorem exists_witness :
∃ i j, i < j ∧ sub (Higman's_mbs i) (Higman's_mbs j) := ...
```

Theorem 3.20 contradicts the fact that *Higman-mbs* is bad, which has been proved at the very beginning of this section. Therefore, we conclude that the only assumption made in this section, that there exists a bad sequence of elements of Q^*, is false.

```
theorem Higman's_contradiction : false :=
badness_of_Higman's_mbs exists_witness
```

Therefore, we conclude that Q^* is wqo.

```
variable {Q : Type}
variable [wqo Q]
-- every sequence over Q* is good
theorem good_star : ∀ f : ℕ → finite_subsets Q , is_good f sub :=
by_contradiction
(suppose ¬ ∀ f, is_good f sub,
have ∃ f, ¬ is_good f sub, from classical.exists_not_of_not_forall this,
Higman's_contradiction this)
-- Q* is good
def wqo_finite_subsets : wqo (finite_subsets Q) :=
⟨⟨⟨sub⟩,sub_refl,sub_trans⟩,good_star⟩
```

3.3 Kruskal's Tree Theorem

3.3.1 A Proof Sketch

We first give a sketch of the proof. The strategy is similar to the one we used to prove Higman's lemma.

Theorem 3.21. *There is no bad sequence of finite trees.*

Proof. (sketch) Suppose that there exists a bad sequence. Then we can construct a minimal bad sequence $mbs\text{-}finite\text{-}tree$ of finite trees. Now we define a new sequence C such that $C(n)$ is the finite set of branches of $mbs\text{-}finite\text{-}tree(n)$. We further define a set

$$mbs\text{-}tree = \bigcup_{i \in \mathbb{N}} C(i)$$

We claim that $mbs\text{-}tree$ is wqo by applying Theorem 3.14. By Higman's lemma, $mbs\text{-}tree^*$ is wqo. Since C is an infinite sequence of elements of $mbs\text{-}tree^*$, C is good. Then there exist i, j such that $i < j$ and $C(i) \leq C(j)$, which subsequently implies that there is an injective and non-descending function f from $C(i)$ to $C(j)$. But this just says that $mbs\text{-}finite\text{-}tree(i) \preceq_{emb} mbs\text{-}finite\text{-}tree(j)$. Therefore, $mbs\text{-}finite\text{-}tree$ is good as witnessed by f. Contradiction. □

```
section
parameter H : ∃ f, ¬ is_good f embeds
-- the minimal bad sequence of finite trees
def mbs_of_finite_tree := minimal_bad_seq size H
-- the sequence C
def seq_branches_of_mbs_tree (n : ℕ) : set (finite_tree × ℕ) :=
branches (mbs_of_finite_tree n)
-- the type mbs_tree
def mbs_tree : Type :=
{t : finite_tree × ℕ // ∃ i, t ∈ seq_branches_of_mbs_tree i}
-- the embedding on mbs_tree
def embeds' (t : mbs_tree) (s : mbs_tree) : Prop := t.val.1 ≼ s.val.1
...
end
```

The subtlety of the formalization of this proof lies in the last step. From the perspective of daily mathematics, it is straightforward that f witnesses the goodness of $mbs\text{-}finite\text{-}tree$. However, by our construction, $mbs\text{-}tree$ is a type. Therefore, f is a function of type $mbs\text{-}tree \to mbs\text{-}tree$. To

show formally that $mbs\text{-}finite\text{-}tree(i) \leq mbs\text{-}finite\text{-}tree(j)$, f should be a function of type $fin\ n \to fin\ m$ for some n and m. We will describe how to obtain the target function from f.

3.3.2 Applying the Mininal Bad Sequence Argument

In the following proofs, we fix the measure $|\cdot|$ to to be the *size* defined on finite trees. Now suppose that there is a bad sequence R of elements of *mbs-tree*. To get a contradiction, it suffices to construct a bad g and a function h such that Hh, H and Hbp are satisfied. By the construction of *mbs-tree*, each element of R must come from some $C(i)$. We define

$$family\text{-}index(n) = \text{an } i \text{ such that } R(n) \in C(i)$$

$$least\text{-}family\text{-}index = \text{the least } l \in ran(family\text{-}index)$$

$$least\text{-}index = \text{a } l \text{ such that } family\text{-}index(l) = least\text{-}family\text{-}index$$

$$Kruskal\text{-}h(n) = family\text{-}index(least\text{-}index + n)$$

$$Kruskal\text{-}g(n) = R(least\text{-}index + n)$$

```
section
parameter H' : ∃ f, ¬ is_good f embeds'
def R : ℕ → mbs_tree := some H'
def family_index (n : ℕ) : ℕ := some ((R n).2)
-- to apply the function least, we have to show that the set is not empty
def index_set_of_mbs_tree : set ℕ := image family_index  univ
lemma index_ne_empty : index_set_of_mbs_tree ≠ ∅ := ...

def least_family_index := least index_set_of_mbs_tree index_ne_empty

lemma exists_least : ∃ i, family_index i = least_family_index :=
have least_family_index ∈ index_set_of_mbs_tree,
  from least_is_mem index_set_of_mbs_tree index_ne_empty,
let ⟨i,h⟩ := this in ⟨i, h.right⟩

def least_index : ℕ := some exists_least

def Kruskal's_g (n : ℕ) : mbs_tree := R (least_index + n)

def Kruskal's_h (n : ℕ) : ℕ :=  family_index (least_index + n)
...
end
```

Theorem 3.22. *Kruskal-g is bad.*

Proof. If it is not bad, then R is not bad. Contradiction. □

```
theorem bad_Kruskal's_g : ¬ is_good Kruskal's_g embeds' :=
suppose is_good Kruskal's_g embeds',
let ⟨i,j,hij⟩ := this in
have least_index + i < least_index + j,
  from add_lt_add_left hij.left _,
have is_good R embeds',
  from ⟨least_index + i, ⟨least_index + j,⟨this, hij.right⟩⟩⟩,
(some_spec H') this
```

Theorem 3.23. *For every i, Kruskal-$h(0) \leq$ Kruskal-$h(i)$.*

Proof. By definition, we have

$$\textit{Kruskal-}h(0) = \textit{least-family-index} \leq \textit{Kruskal-}h(i)$$

□

```
theorem Kruskal's_Hh (n : ℕ) : Kruskal's_h 0 ≤ Kruskal's_h n :=
have Kruskal's_h 0 = family_index least_index, from rfl,
have family_index least_index = least_family_index,
  from some_spec exists_least,
have Kruskal's_h 0 = least_family_index, by simph,
by rw this; apply minimality; apply family_index_in_index_of_mbs_tree
```

Theorem 3.24. $|\textit{Kruskal-}g(0)| < |\textit{mbs-finite-tree}(\textit{Kruskal-}h(0))|$

Proof. It suffices to show that

$$|R(\textit{least-index})| < |\textit{mbs-finite-tree}(\textit{least-family-index})|$$

Since $R(\textit{least-index}) \in C(\textit{least-family-index})$, $R(\textit{least-index})$ is a branch of $\textit{mbs-finite-tree}(\textit{least-family-index})$. By Theorem 2.3, we have the result. □

```
theorem size_elt_Kruskal's_g_lt_mbs_finite_tree (n : ℕ) :
size (Kruskal's_g n).val.1 < size (mbs_of_finite_tree (Kruskal's_h n)) :=
lt_of_size_of_branches (some_spec (Kruskal's_g n).2)
-- a special case of the above theorem
theorem Kruskal's_Hbp :
size (Kruskal's_g 0).val.1 < size (mbs_of_finite_tree (Kruskal's_h 0)) :=
size_elt_Kruskal's_g_lt_mbs_finite_tree 0
```

Theorem 3.25. *For every i, j, if mbs-finite-tree$(i) \preceq_{emb}$ Kruskal-g(j), then mbs-finite-tree$(i) \preceq_{emb}$ mbs-finite-tree(Kruskal-h(j)).*

Proof. By definition,

$$R(\text{least-index} + j) \in C(\text{family-index}(\text{least-index} + j))$$

We have

$$\text{Kruskal-g}(j) = R(\text{least-index} + j) \in C(\text{Kruskal-h}(j))$$

meaning that Kruskal-g(j) is a branch of mbs-finite-tree(Kruskal-h(j)). By Theorem 2.2, Kruskal-g$(j) \preceq_{emb}$ mbs-finite-tree(Kruskal-h(j)). By transitivity of homeomorphic embedding, we have the result. □

```
theorem trans_of_Kruskal's_g {i j : ℕ}
(H1 : mbs_of_finite_tree i ≼ (Kruskal's_g j).val.1) :
mbs_of_finite_tree i ≼ mbs_of_finite_tree (Kruskal's_h j) :=
have (Kruskal's_g j).val ∈ branches (mbs_of_finite_tree (Kruskal's_h j)),
  from some_spec (Kruskal's_g j).2,
have (Kruskal's_g j).val.1 ≼ mbs_of_finite_tree (Kruskal's_h j),
  from embeds_of_branches this,
embeds_trans H1 this
```

Theorem 3.26. *For every i, j, if*

$$\text{mbs-finite-tree}(i) \preceq_{emb} \text{Kruskal-g}(j - \text{Kruskal-h}(0))$$

then

$$\text{mbs-finite-tree}(i) \preceq_{emb} \text{mbs-finite-tree}(\text{Kruskal-h}(j - \text{Kruskal-h}(0)))$$

Proof. Immediately follows from Theorem 3.25. □

Theorem 3.27. *There is a contradiction.*

Proof. Consider the hypothetical constants in the context of Theorem 3.14. Let g be Kruskal-g and h be Kruskal-h. Let Hh be Theorem 3.23, Hbp be Theorem 3.24 and H be Theorem 3.26. By Theorem 3.14, we have the contradiction. □

```
-- a one-line proof
theorem Kruskal's_local_contradiction : false :=
local_contra_of_comb_seq_with_mbs ...
```

Therefore, we conclude that *mbs-tree* is wqo. By Higman's lemma, *mbs-tree** is wqo. Then there is an injective and non-descending f which witnesses that $C(i) \leq C(j)$ for some $i < j$. We should be able to conclude from the fact that *mbs-finite-tree* is good, which gives us a contradiction to conclude that *finite-tree* is wqo. The problem is that f is not of the correct type. We now define a function *recover* which extracts a funtion of the correct type from f. Our mathematical presentation ends here. Since the following is all about type-theoretic encoding and transformation, we give explanations of the code directly rather than present the transformation in mathematical language.

```
theorem embeds'_is_good : ∀ f, is_good f embeds' :=
by_contradiction
(suppose ¬ ∀ f, is_good f embeds',
 have ∃ f, ¬ is_good f embeds',
   from classical.exists_not_of_not_forall this,
 Kruskal's_local_contradiction this)
-- mbs_tree is wqo
instance wqo_mbs_tree : wqo mbs_tree :=
wqo.mk (quasiorder.mk (has_le.mk embeds') embeds'_refl embeds'_trans)
    embeds'_is_good
-- mbs_tree* is wqo
def wqo_finite_subsets_of_mbs_tree : wqo (finite_subsets mbs_tree) :=
wqo_finite_subsets
```

3.3.3 Type-theoretic Transformations

What we have is the claim that *mbs-tree** is wqo. Recall that *mbs-tree* is the collection of all the branches appearing in the minimal bad sequence of finite trees. On the other hand, C is a sequence of sets of finite trees, i.e., C is a function of type $N \to$ *set* (*finite-tree* $\times \mathbb{N}$). We cannot conclude immediately that C is good because *mbs-tree** and *set* (*finite-tree* $\times \mathbb{N}$) are two different types. We first construct a "mirror" of C so that the claim can be applied.

```
-- extract the ordering on mbs_tree* from the claim
def os : finite_subsets mbs_tree → finite_subsets mbs_tree → Prop :=
wqo_finite_subsets_of_mbs_tree.le
-- our claim ensures that this ordering is good
theorem good_finite_subsets_of_mbs_tree : ∀ f, is_good f os :=
wqo_finite_subsets_of_mbs_tree.is_good
-- branches of mbs_of_finite_tree form a set of mbs_tree
def elt_mirror (n : ℕ) : set mbs_tree :=
{x : mbs_tree | x.1 ∈ seq_branches_of_mbs_tree n}
```

```
-- the mirror should be faithful, i.e., everything in it is in C and vice versa
theorem mirror_refl_left (x : mbs_tree) (n : ℕ) :
x ∈ elt_mirror n → x.1 ∈ seq_branches_of_mbs_tree n := λ Hx, Hx
theorem mirror_refl_right (x : mbs_tree) (n : ℕ) :
x.1 ∈ seq_branches_of_mbs_tree n → x ∈ elt_mirror n := λ Hx, Hx
-- each element of C is finite
instance  finite_seq_branches (n : ℕ) :
finite (seq_branches_of_mbs_tree n) :=
finite_branches (mbs_of_finite_tree n)
-- each element of the mirror is finite
theorem finite_elt (n : ℕ) : finite (elt_mirror n) :=
have mapsto : maps_to subtype.val (elt_mirror n)
  (seq_branches_of_mbs_tree n), from λ x Hx, mirror_refl_left x n Hx,
have inj_on subtype.val (elt_mirror n), from λ x₁ x₂ H₁ H₂, subtype.eq,
finite_of_inj_on mapsto this
-- intuitively, the mirror is a reflection of C, but is of the type mbs_tree*
def mirror (n : ℕ) : finite_subsets mbs_tree :=
⟨elt_mirror n, finite_elt n⟩
```

Next we check that the mirror is good. This is an immediate consequence of the claim we have.

```
theorem good_mirror : ∃ i j, i < j ∧ os (mirror i) (mirror j) :=
good_finite_subsets_of_mbs_tree mirror
```

Now we have an injective and non-descending $f : \mathit{mbs\text{-}tree} \to \mathit{mbs\text{-}tree}$ from $\mathit{mirror}(i)$ to $\mathit{mirror}(j)$. Recall that our goal is to show that there exists a function g which witnessess that

$$\mathit{mbs\text{-}finite\text{-}tree}(i) \preceq_{emb} \mathit{mbs\text{-}finite\text{-}tree}(j)$$

For the convenience of description, we destruct the claim *good-mirror* to make all the available facts manageable.

```
section
-- we want to show that there exist i, j such that (mbs_of_finite_tree i ≼
    mbs_of_finite_tree j)
-- fortunately, i and j from good_mirror suffice
parameters {i j : ℕ}
-- since the mirror is good, we have an injective and non−descending f'
parameter f' : mbs_tree → mbs_tree
parameter inj : inj_from_to f' (elt_mirror i) (elt_mirror j)
parameter nond : ∀ a : mbs_tree, a.val ∈ seq_branches_of_mbs_tree i →
          a.val.1 ≼ (f' a).val.1 ∧  (f' a).val ∈ seq_branches_of_mbs_tree j
...
```

```
end
```

Now further assume that *mbs-finite-tree*(i) is of the form *cons tsi* where $tsi : fin\ ni \to finite\text{-}tree$ for some ni and *mbs-finite-tree*(j) is of the form *cons tsj* where $tsj : fin\ nj \to finite\text{-}tree$ for some nj. We also know that for every a, $(tsi\ a, val\ a) \in B(mbs\text{-}finite\text{-}tree(i))$ by definition.

```
-- suppose (mbs_of_finite_tree i) is of the form (cons tsi)
-- suppose (mbs_of_finite_tree j) is of the form (cons tsj)
parameters ni nj : ℕ
parameters (tsi : fin ni → finite_tree) (tsj : fin nj → finite_tree)
-- suppose we know that they are destructed
parameter eqi : mbs_of_finite_tree i = cons tsi
parameter eqj : mbs_of_finite_tree j = cons tsj
-- recall that each branch of (mbs_of_finite_tree i) is in C(i)
parameter Htsi : ∀ a, (tsi a, val a) ∈ seq_branches_of_mbs_tree i
```

Next we define a function *mbst-form*, which takes a $ti : fin\ ni$ and returns a *mbs-tree*. The intuition is that *mbst-form*(ti) is the *mbs-tree* form of $tsi(ti)$.

```
-- every ti corresponds to some C(i)
lemma foo (ti : fin ni) : ∃ i, (tsi ti, val ti) ∈ seq_branches_of_mbs_tree i
:= ⟨i,Htsi ti⟩
-- given a ti, find the corresponding mbs_tree of (tsi ti).
def mbst_form (ti : fin ni) : mbs_tree := ⟨(tsi ti, val ti),(foo ti)⟩
```

We can show that for every a, *mbst-form*(a) is in *mirror*(a).

```
theorem mem_mbst_form (a : fin ni) : mbst_form a ∈ elt_mirror i := Htsi a
```

The function *mbst-form* is injective because it always keeps a copy of the value of the input.

```
theorem eq_of_mbst_form {a₁ a₂ : fin ni}
(Heq : mbst_form a₁ = mbst_form a₂) : a₁ = a₂
:= by apply eq_of_veq;super
```

With these hypothetical constants and auxiliary functions, we decribe the behavior of the function *recover*. Given a $ti : fin\ ni$, *recover* does the following.

1. Find the *mbst-form* of ti, say x.
2. By *nond*, we know that $f(x).val$ is in $B(mbs\text{-}finite\text{-}tree(j))$.
3. This means that some $a : fin\ nj$ corresponds to ti.
4. By axiom of choice, take such an $a : fin\ nj$.

```
def recover (ti : fin ni) : fin nj :=
have (f' (mbst_form ti)).val ∈ seq_branches_of_mbs_tree j,
  from (nond _ (begin dsimp [mbst_form], apply Htsi end)).right,
have mem : (f' (mbst_form ti)).val ∈ branches (mbs_of_finite_tree j),
  from this,
have branches (mbs_of_finite_tree j) = branches (cons tsj), by rw eqj,
have (f' (mbst_form ti)).val ∈ branches_aux tsj,
  by rw this at mem;exact mem,
some this
```

We check that *recover* does have the properties we want, i.e., it is a proper witness of (E2).

```
-- the second clause of E2
theorem perm_recover (ti : fin ni) : tsi ti ≼ tsj (recover ti) :=
have (f' (mbst_form ti)).val ∈ seq_branches_of_mbs_tree j,
  from (nond _ (begin dsimp [mbst_form], apply Htsi end)).right,
have mem : (f' (mbst_form ti)).val ∈ branches (mbs_of_finite_tree j),
  from this,
have branches (mbs_of_finite_tree j) = branches (cons tsj), by rw eqj,
have (f' (mbst_form ti)).val ∈ branches_aux tsj,
  by rw this at mem;exact mem,
have tsj (recover ti) = (f' (mbst_form ti)).val.1,
  from let ⟨a,b⟩ := some_spec this in a,
have tsi ti ≼ (f' (mbst_form ti)).val.1,
  from (nond _ (begin dsimp [mbst_form], apply Htsi end)).left,
by simph
-- the first clause of E2
theorem inj_recover : injective recover :=
λ a₁ a₂ Heq,
have (f' (mbst_form a₁)).val ∈ seq_branches_of_mbs_tree j,
  from (nond _ (begin dsimp [mbst_form], apply Htsi end)).right,
have mem : (f' (mbst_form a₁)).val ∈ branches (mbs_of_finite_tree j),
  from this,
have branches (mbs_of_finite_tree j) = branches (cons tsj), by rw eqj,
have (f' (mbst_form a₁)).val ∈ branches_aux tsj,
  by rw this at mem;exact mem,
have eeq1 : ∃ a : fin nj, tsj a = (f' (mbst_form a₁)).val.1 ∧
  val a = (f' (mbst_form a₁)).val.2, from this,
have pr11 : tsj (recover a₁) = (f' (mbst_form a₁)).val.1,
  from let ⟨a,b⟩ := some_spec eeq1 in a,
have pr21 : val (recover a₁) = (f' (mbst_form a₁)).val.2,
  from let ⟨a,b⟩ := some_spec eeq1 in b,
```

```
have (f' (mbst_form a₂)).val ∈ seq_branches_of_mbs_tree j,
  from (nond _ (begin dsimp [mbst_form], apply Htsi end)).right,
have mem : (f' (mbst_form a₂)).val ∈ branches (mbs_of_finite_tree j),
  from this,
have branches (mbs_of_finite_tree j) = branches (cons tsj), by rw eqj,
have (f' (mbst_form a₂)).val ∈ branches_aux tsj,
  by rw this at mem;exact mem,
have eeq2 : ∃ a : fin nj, tsj a = (f' (mbst_form a₂)).val.1 ∧
  val a = (f' (mbst_form a₂)).val.2, from this,
have pr12 : tsj (recover a₂) = (f' (mbst_form a₂)).val.1,
  from let ⟨a,b⟩ := some_spec eeq2 in a,
have pr22 : val (recover a₂) = (f' (mbst_form a₂)).val.2,
  from let ⟨a,b⟩ := some_spec eeq2 in b,
have eq1 : (f' (mbst_form a₁)).val.1 = (f' (mbst_form a₂)).val.1,
  by rw [-pr12, -pr11, Heq],
have (f' (mbst_form a₁)).val.2 = (f' (mbst_form a₂)).val.2,
  by rw [-pr22,-pr21, Heq],
have (f' (mbst_form a₁)).val = (f' (mbst_form a₂)).val,
  from eq_of_prod eq1 this,
have f'eq : f' (mbst_form a₁) = f' (mbst_form a₂),
  from subtype.eq this,
have ∀ x₁ x₂ : mbs_tree, x₁ ∈ elt_mirror i →
  x₂ ∈ elt_mirror i → f' x₁ = f' x₂ → x₁ = x₂, from and.right inj,
have mbst_form a₁ = mbst_form a₂, from this (mbst_form a₁) (mbst_form a₂)
  (mem_mbst_form a₁) (mem_mbst_form a₂) f'eq,
show _, from eq_of_mbst_form this
```

Using *recover*, we extract a function of the correct type that witnesses that *mbs-finite-tree* is good. This fact gives us a contradiction to conclude further that *finite-tree* is well quasi-ordered under homeomorphic embedding.

```
theorem good_mbs_of_finite_tree :
∃ i j, i < j ∧ mbs_of_finite_tree i ≼ mbs_of_finite_tree j :=
let ⟨i,j,⟨iltj,⟨f',⟨inj,nond⟩⟩⟩⟩ := good_mirror in
let ⟨ni,tsi,htsi⟩ := @finite_tree_destruct (mbs_of_finite_tree i) in
let ⟨nj,tsj,htsj⟩ := @finite_tree_destruct (mbs_of_finite_tree j) in
have Htsi : ∀ a, (tsi a, val a) ∈ seq_branches_of_mbs_tree i,
  from take a, mem_of_seq_branches tsi a htsi,
let f (a : fin ni) : fin nj := recover f' nond ni _ tsi tsj htsj Htsi a in
have injf : injective f, from inj_recover _ inj _ _ _ _ _ _ _,
have ∀ z : fin ni, tsi z ≼ tsj (f z), from λ z, perm_recover _ _ _ _ _ _ _ _ _,
have cons tsi ≼ cons tsj, from or.inr ⟨f,⟨injf,this⟩⟩,
have mbs_of_finite_tree i ≼ mbs_of_finite_tree j, by simph,
```

```
⟨i,j,⟨iltj,this⟩⟩
-- a contradiction is obatained as we know taht the minimal bad sequence is bad
theorem Kruskal's_contradiction : false :=
bad_mbs_finite_tree good_mbs_of_finite_tree
-- we conclude that our assumption must be false
theorem embeds_is_good : ∀ f, is_good f embeds :=
by_contradiction
(suppose ¬ ∀ f, is_good f embeds,
 have ∃ f, ¬ is_good f embeds,
 from classical.exists_not_of_not_forall this,
 Kruskal's_contradiction this)
-- the set of finite trees is wqo
def wqo_finite_tree : wqo finite_tree :=
⟨⟨⟨embeds⟩,embeds_refl,@embeds_trans⟩,embeds_is_good⟩
```

4 Conclusion and Future Work

In this thesis, we have formalized the classical proof of Kruskal's tree theorem given by Nash-Williams. This is the first formalization of the theorem in Lean. Along the way, a general abstraction of the minimal bad sequence argument is developed so that the argument can be applied to any proof as long as suitable instances of assumptions are constructed. Moreover, the abstraction incorporates not only the construction of the minimal bad sequence, but also the proof of a subsequent contradiction. This shortens the formalization significantly because we do not have to formalize two proofs with similar proof strategies twice. For the tree theorem, we used a simpler type-theoretic encoding of finite trees and homeomorphic embedding which, we believe, encoded a clearer intuition of finite trees and embeddability.

Future work includes a formalization of the theorem using an extended notion of finite trees so that finite trees can be built on a set of labels, and a formalization of Higman's lemma in terms of lists. Essential tools for formalizing them have already been developed in the current formalization so we expect that they can be completed in the near future[3]. With the list version of Higman's lemma and a suitable encoding of finite trees in terms of lists, the type-theoretic transformation in section 3.3.3 might be avoidable. Other optimizations include using more automation on linear arithmetic (to be implemented in Lean) to simplify the proofs that require substantial reasoning in arithmetic, and translating some definitions into their equivalent froms so that some theorems have more elegant proofs. A new objective is to formalize an alternative version of Kruskal's tree theorem found in J.J. Levy's unpublished notes, as reported in [5], which says that the set of finite trees is *wqo* under homeomorphic embedding if it is *wqo* under certain given quasi-order. We hope that our work will finally gather enough basic tools in the WQO theory so that future formal development of the theory, as well as proving termination in Lean becomes possible.

[3]In fact, the list version of Higman's lemma has been formalized right after finishing this thesis. See the GitHub repository. `https://github.com/minchaowu/Kruskal.lean3`

References

[1] Leonardo De Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *arXiv preprint arXiv:1505.04324*, 2015.

[2] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer International Publishing, 2015.

[3] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004.

[4] Harvey M Friedman, Kenneth McAloon, and Stephen G Simpson. A finite combinatorial principle which is equivalent to the 1-consistency of predicative analysis. *Studies in Logic and the Foundations of Mathematics*, 109:197–230, 1982.

[5] Jean H Gallier. What's so special about Kruskal's theorem and the ordinal Γ_0? A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991.

[6] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326–336, 1952.

[7] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[8] Joseph B Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.

[9] C St JA Nash-Williams. On well-quasi-ordering finite trees. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 59, pages 833–835. Cambridge Univ Press, 1963.

[10] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[11] Mitsuhiro Okada and Gaisi Takeuti. On the theory of quasi-ordinal diagrams. *Logic and Combinatorics. American Mathematical Society*, 1986.

[12] Monika Seisenberger. Kruskal's tree theorem in a constructive theory of inductive definitions. In *Reuniting the Antipodes-Constructive and Nonstandard Views of the Continuum*, pages 241–255. Springer, 2001.

[13] Stephen G Simpson. Nonprovability of certain combinatorial properties of finite trees. *Studies in Logic and the Foundations of Mathematics*, 117:87–117, 1985.

[14] D Singh, Ali Mainguwa Shuaibu, and MS Ndayawo. Simplified proof of Kruskal's tree theorem. 2013.

[15] Christian Sternagel. Certified Kruskal's tree theorem. In *3rd International Conference on Certified Programs and Proofs (CPP)*, pages 178–193. Springer International Publishing, 2013.