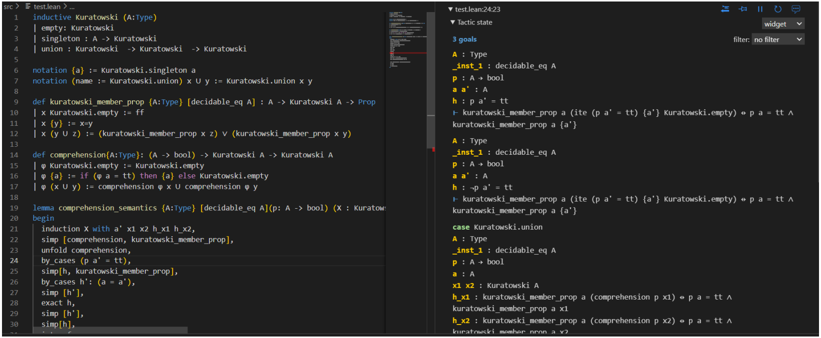
Formalization of finite sets in Lean

Johannes Tantow

August 14, 2023

Formalization of finite sets **in Lean**



src > test.lean > ...

```lean
inductive Kuratowski (A:Type)
| empty: Kuratowski
| singleton : A -> Kuratowski
| union : Kuratowski  -> Kuratowski  -> Kuratowski

notation {a} := Kuratowski.singleton a
notation (name := Kuratowski.union) x ∪ y := Kuratowski.union x y

def kuratowski_member_prop {A:Type} [decidable_eq A] : A -> Kuratowski A -> Prop
| x Kuratowski.empty := ff
| x {y} := x=y
| x (y ∪ z) := (kuratowski_member_prop x z) ∨ (kuratowski_member_prop x y)

def comprehension{A:Type}: (A -> bool) -> Kuratowski A -> Kuratowski A
| φ Kuratowski.empty := Kuratowski.empty
| φ {a} := if (φ a = tt) then {a} else Kuratowski.empty
| φ (x ∪ y) := comprehension φ x ∪ comprehension φ y

lemma comprehension_semantics {A:Type} [decidable_eq A](p: A -> bool) (X : Kuratows
begin
  induction X with a' x1 x2 h_x1 h_x2,
  simp [comprehension, kuratowski_member_prop],
  unfold comprehension,
  by_cases (p a' = tt),
  simp[h, kuratowski_member_prop],
  by_cases h': (a = a'),
  simp [h'],
  exact h,
  simp [h'],
  simp[h],
```

▼ test.lean:24:23

▼ Tactic state

widget

3 goals

filter: no filter

```
A : Type
_inst_1 : decidable_eq A
p : A → bool
a a' : A
h : p a' = tt
⊢ kuratowski_member_prop a (ite (p a' = tt) {a'} Kuratowski.empty) ↔ p a = tt ∧
kuratowski_member_prop a {a'}

A : Type
_inst_1 : decidable_eq A
p : A → bool
a a' : A
h : ¬p a' = tt
⊢ kuratowski_member_prop a (ite (p a' = tt) {a'} Kuratowski.empty) ↔ p a = tt ∧
kuratowski_member_prop a {a'}

case Kuratowski.union
A : Type
_inst_1 : decidable_eq A
p : A → bool
a : A
x1 x2 : Kuratowski A
h_x1 : kuratowski_member_prop a (comprehension p x1) ↔ p a = tt ∧
kuratowski_member_prop a x1
h_x2 : kuratowski_member_prop a (comprehension p x2) ↔ p a = tt ∧
kuratowski_member_prop a x2
```

Formalization of finite **sets** in Lean

Naive set theory

A set is collections of objects.

Zermelo-Fraenkel set theory:

1. Axiom of Extensionality: $\forall x, y.(\forall z.(z \in x \leftrightarrow z \in y) \to x = y)$
2. Axiom of Regularity $\forall x.(x \neq \emptyset \to \exists y.y \in x \wedge y \cap x = \emptyset)$
3. Axiom of Empty Set: $\exists y.\forall x : \neg x \in y$
4. ...

Formalization of finite **sets** in Lean

Naive set theory

A set is collections of objects.

Zermelo-Fraenkel set theory:

1. Axiom of Extensionality: $\forall x, y.(\forall z.(z \in x \leftrightarrow z \in y) \rightarrow x = y)$
2. Axiom of Regularity $\forall x.(x \neq \emptyset \rightarrow \exists y.y \in x \wedge y \cap x = \emptyset)$
3. Axiom of Empty Set: $\exists y.\forall x : \neg x \in y$
4. ...

Lean

$(s : \text{set } A) := A \rightarrow \text{Prop}$

Formalization of **finite sets** in Lean

1. there is a list containing all elements

Formalization of **finite sets** in Lean

1. there is a list containing all elements
2. there is a tree containing all elements

Formalization of **finite sets** in Lean

1. there is a list containing all elements
2. there is a tree containing all elements
3. there exists some $N \in \mathbb{N}$ such that every list of length at least N contains duplicates

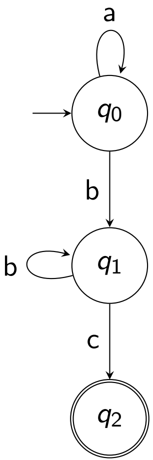
Formalization of **finite sets** in Lean

1. there is a list containing all elements
2. there is a tree containing all elements
3. there exists some $N \in \mathbb{N}$ such that every list of length at least N contains duplicates
4. there is no surjection from this set to the natural numbers

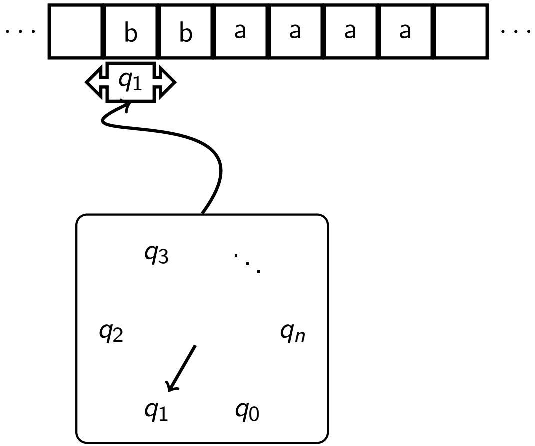
Formalization of **finite sets** in Lean

1. there is a list containing all elements
2. there is a tree containing all elements
3. there exists some $N \in \mathbb{N}$ such that every list of length at least N contains duplicates
4. there is no surjection from this set to the natural numbers
5. ...

Finite sets in automata theory



$(\mathbf{Q}, s, \boldsymbol{\Sigma}, F, \delta)$



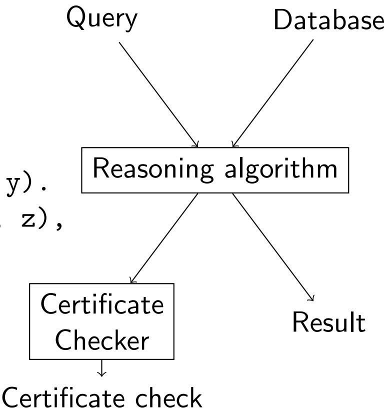
$(\mathbf{Q}, \boldsymbol{\Sigma}, \boldsymbol{\Gamma}, \delta, q_0, \sqcup, \mathbf{F})$

Finite sets in databases

Trains

Train	Start	Stop
RE3	Stralsund	Berlin
RE4	Stendal	Berlin

```
Reachable(x,y) :- Trains(t, x, y).
Reachable(x,y) :- Reachable (x, z),
                  Trains(z,y).
```



Goals

- Implement different versions of the same results
- set operations like $\cup, \cap, \setminus$ or *size*
- Notions of equality

Desired result

$$size(X \cup Y) + size(X \cap Y) = size(X) + size(Y)$$

- Correctness of implementation
- Requirements
- useabilty in computation
- easyness of proofs
- availability of induction
- support from the standard library

Formalization of **finite sets in Lean** - so far

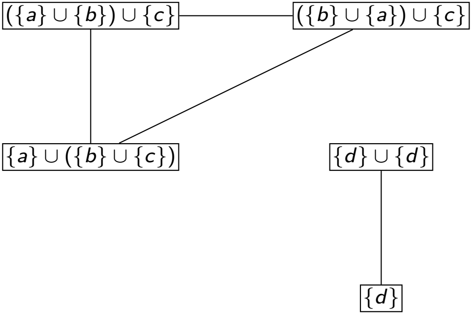
```
structure finset (α : Type*) :=
    (val : multiset α)
    (nodup : nodup val)


section lattice
    variables (α : Type*) [decidable_eq α]
    instance : has_union (finset α)
    instance : has_inter (finset α)

    ...
```

HIT from HoTT

Types as Space



Kuratowski sets in Lean

```
inductive K (A:Type u)
| empty: K
| singleton : A -> K
| union : K  -> K  -> K
```

Kuratowski sets in Lean

```
inductive K (A:Type u)
| empty: K
| singleton : A -> K
| union : K  -> K  -> K
```

```
axiom union_comm (x y : K A): x ∪ y = y ∪ x
axiom union_singleton_idem  (x : A): {x} ∪ {x} = {x}
axiom union_assoc (x y z : K A):
        x ∪ (y ∪ z) = (x ∪ y) ∪ z
axiom empty_union (x : K A): empty ∪ x = x
axiom union_empty {x: K A} : x ∪ empty = x
```

Size

```
def size{A:Type}: KSet A -> A
| ∅   := 0
| {a} := 1
| (X ∪ Y) :=  ?
```

Size

```
def size{A:Type}: KSet A -> A
| ∅   := 0
| {a} := 1
| (X ∪ Y) :=  ?
```

Proposal

$size(X \cup Y) := size(X) + size(Y) - size(X \cap Y)$

Size problems

$$(\{1\} \cup \{2\}) \cup (\{2\} \cup \{3\})$$
$$\{1\} \cup (\{2\} \cup (\{2\} \cup \{3\}))$$

Size problems

$$(\{1\} \cup \{2\}) \cup (\{2\} \cup \{3\})$$
$$\{1\} \cup (\{2\} \cup (\{2\} \cup \{3\}))$$

```
def to_list: Kuratowski A -> list A
| ∅   := nil
| {a} := a :: nil
| (X ∪ Y) := to_list X ++ to_list y

def size (X: Kuratowski A): ℕ := len(to_list(X))
```

Axioms and Functions

```
axiom union_comm (X Y: KSet A): X ∪ Y = Y ∪ X

noncomputable def first{A:Type} [nonempty A]: KSet A
    -> A
| ∅   := classical.some A
| {a} := a
| (X ∪ Y) := first (X)
```

Axioms don't care about preservation by functions

One million dollars

```
theorem p_eq_np (l: language)(l_np: l ∈ NP):l ∈ P :=
begin
    exfalso,
    let X:= {1} ∪ {2},
    have first1: first (X) = 1,
    dunfold first,
    refl,
    have first2: ¬  first(X) = 1,
    have X2: X = {2} ∪ {1} by union_comm,
    rw X2,
    simp[first],
    exact absurd first1 first2,
end
```

Quotients

Definition

Let A : Type and $R : A \times A \to$ Prop be an equivalence relation. Then A/R, the quotient of A by R, is a type.

Quotients

Definition

Let A : Type and $R : A \times A \to \text{Prop}$ be an equivalence relation. Then A/R, the quotient of A by R, is a type.

Equivalence relations:

1. $R(X, Y) := (X = Y) \vee (\exists V, W.X = (V \cup W) \wedge Y = (W \cup Y)) \vee \ldots$

Quotients

Definition

Let A : Type and $R : A \times A \to$ Prop be an equivalence relation. Then A/R, the quotient of A by R, is a type.

Equivalence relations:

1. $R(X, Y) := (X = Y) \vee (\exists V, W.X = (V \cup W) \wedge Y = (W \cup Y)) \vee \ldots$
2. $R(X, Y) := \forall a.a \in X \leftrightarrow a \in Y$

Lifting

```
def member (a:A) (X: listSet A) :=
    quot.lift list.member member_correctness


lemma member_correctness (l1 l2: list A)(eq: R l1 l2)
     (a:A): member a l1 = member a l2 :=
begin
        unfold R at eq,
        rw eq_as_iff,
        apply eq,
end
```

Trees

```
inductive Tree (A : Type)
| empty: Tree
| node : Tree -> A -> Tree -> Tree


def ordered: Tree A -> Prop
| empty := true
| (node  tl x tr):= ordered tl ∧
                    ordered tr ∧
                    (forall_keys (>) x tl) ∧
                    (forall_keys (<) x tr)
```

Ordered Trees

```
def size: Tree A -> ℕ
| empty := 0
| node := 1 + size tl + size tr

structure ordered_tree (A: Type) [linear_order A] :=
    (base: binaryTree.Tree A)
    (o: binaryTree.ordered base)

def size (T: ordered_tree A): ℕ := size T.base
```

Insertion

```
def unbalanced_insert : A -> Tree A -> Tree A
| x Tree.empty := (Tree.node Tree.empty x Tree.empty)
| x (Tree.node  tl a tr) :=
    if (x = a)
    then (Tree.node tl a tr)
    else if x < a
    then Tree.node (unbalanced_insert x tl) a tr
    else
    Tree.node tl a (unbalanced_insert x tr)
```

Correct insertion

```
lemma member_after_insert (a: A) (t: Tree A): member
    a (unbalanced_insert a t)

lemma insert_keeps_previous_members (t:Tree A) (a b:A
    ):
member a t → member a (unbalanced_insert b t)

lemma insert_only_adds_argument (a b: A) (t: Tree A):
member b (unbalanced_insert a t) → member b t∨(b=a)
```

Correct insertion

```
lemma member_after_insert (a: A) (t: Tree A): member
    a (unbalanced_insert a t)

lemma insert_keeps_previous_members (t:Tree A) (a b:A
    ):
member a t → member a (unbalanced_insert b t)

lemma insert_only_adds_argument (a b: A) (t: Tree A):
member b (unbalanced_insert a t) → member b t∨(b=a)

def union: Tree A -> Tree A -> Tree A
| B Tree.empty := B
| B (Tree.node tl x tr) := union ( union (
    unbalanced_insert x B) tl) tr
```

Difference I

```
def comprehension: (A -> bool) ->  Tree A -> Tree A
| φ Tree.empty := binaryTree.empty
| φ (Tree.node tl x tr) :=  if φ x = tt then union (
    unbalanced_insert x (comprehension φ tl)) (
    comprehension φ tr) else union (comprehension φ
    tl) (comprehension φ tr)

def difference (X Y: Tree A) : Tree A :=
    comprehension (λ (a:A), ¬ (member_bool a Y = tt))
     X
```

Tree equality

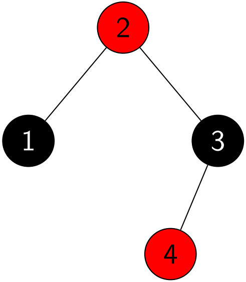


Figure: flatten T = [1,2,3,4]

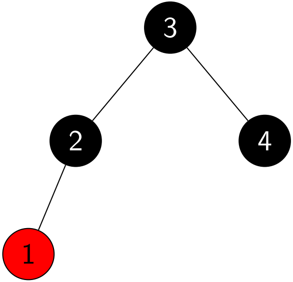


Figure: flatten T = [1,2,3,4]

Use a quotient based on flatten.

Properties of flatten

Lemma(Extensionality)

$flatten(T1) = flatten(T2) \leftrightarrow \forall x, x \in T1 \leftrightarrow x \in T2$

Proof

Idea:

- two list are equal iff they are both sorted and permutations of each other
- Ordered trees are sorted
- Ordered trees are duplicate free: permutations

Lemma

size(T) = len(flatten(T))

Induction on trees

TreeSet:
Use Quot.induction: Show the result for quot.mk T for every tree T

↓

OrderedTree:
Unfold operations on from quotient operations

↓

BinaryTree:
Unfold operations
Induction on trees, orderedness is kept

Don't repeat yourself

Mathlib data.set

```
theorem mem_inter_iff (x : α) (a b : set α) :
    x ∈ a ∩ b ↔ (x ∈ a ∧ x ∈ b)
```

Kuratowski sets

```
lemma in_intersection_iff_in_both (X Y: Kuratowski A)
     (a:A): a ∈ (X ∩ Y) =  (a ∈ X ∧ a ∈ Y)
```

Finite by proof

Definition

A finite set S_f is a pair of a set S and a proof of its finiteness.

Finite by proof

Definition

A finite set S_f is a pair of a set S and a proof of its finiteness.

Examples

1. Bijection finite:
   $\exists (n : \mathbb{N}), (f : \mathbb{N} \to A).\ \text{set.bij\_on}\ f\ \{x \in \mathbb{N} \mid x < n\}\ S$
2. Surjection finite:
   $\exists (n : \mathbb{N}), (f : \mathbb{N} \to A).\ \text{set.surj\_on}\ f\ \{x \in \mathbb{N} \mid x < n\}\ S$
3. Dedekind finite: $\forall (S' \subsetneq S), (f : A \to A).\ \neg \text{set.bij\_on}\ f\ S'\ S$

Simple sets

Lemma

$\varnothing$ is finite.

$f : n \mapsto classical.some\ A$

Lemma

For all $a : A$, $\{a\}$ is finite.

$f : n \mapsto a$

Type Requirement

A has to be nonempty

Size

```
def is_finite {A: Type} (S: set A): Prop :=
    ∃ (n:ℕ) (f: ℕ → A),
    set.bij_on f (set_of (λ (a:ℕ), a < n)) S
```

How to get n for $\exists n, \phi(n)$?

1. classical.choie: Uses AC
2. nat.find if ϕ is decidable

```
noncomputable def size (s: set A) (fin: is_finite s):
     ℕ
:= classical.some fin
```

Noncomputable size

- Let M be a Turing-machine.
- Then $\{M\}$ is a finite set.
- There exists a *FO* formula $\phi(M)$, that is true whenever a TM stops on the empty input
- Then $\{M' \mid \phi(M') \wedge M' \in \{M\}\}$? is finite.
- What is the size of $\{M' \mid \phi(M') \wedge M' \in \{M\}\}$?

Computable size

```
def set_size: listSet A -> ℕ
| nil := 0
| (hd::tl) := if hd ∈ tl then set_size(tl) else
    set_size(tl) + 1

def singleton: A -> listSet A := {a}
def comprehension: (A -> bool) -> listSet A ->
    listSet A
```

Lean detail

Every function $A \to bool$ is computable, whereas $A \to Prop$ is not.

```
noncomputable def comprehension': (A -> Prop) ->
    listSet A -> listSet A
```

Proving size

$$\texttt{is\_finite\_n}\ (n : \mathbb{N}):$$
$$\exists (f : \mathbb{N} \to A).\ \texttt{set.bij\_on}\ f\ \{x \in \mathbb{N} \mid x < n\}\ S$$

Goal: is_finite_n$(S, n) \leftrightarrow$ size$(S) = n$

```
lemma no_bijection_between_different_lt_n (n1 n2: ℕ)
    (n1_ne_n2: n1 ≠ n2) :
∀ (f: ℕ → ℕ ), ¬ set.bij_on f {x| x < n1} {x| x <
    n2}
```

Intersection, Difference and subsets

```
lemma subset_of_fin_is_fin (s1 s2: set A)
(subs: s2 ⊆  s1) (s1_fin: is_finite s1):
    is_finite s2
```

Proof Idea

Every subset S of $\{x \mid x < n\}$ is bijective to $\{x \mid x < m\}$ for some m.

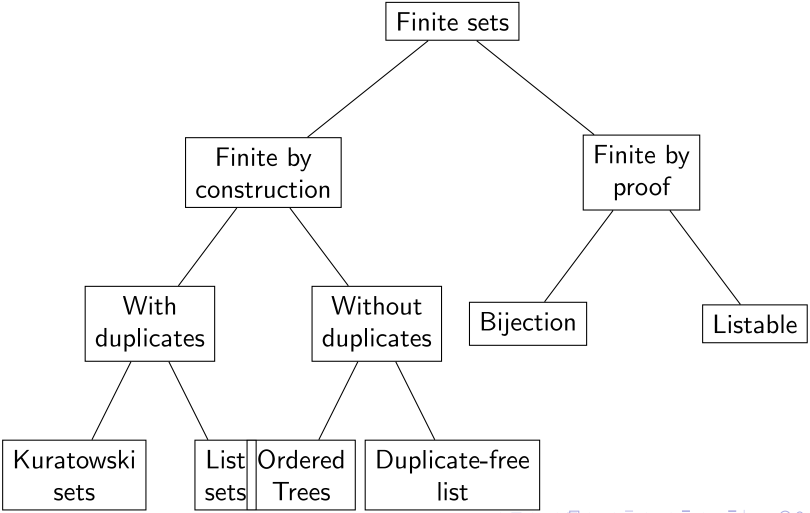
Induction on n $n = 0$: Empty set is finite.

$n = m + 1$: Case distinction: $m \in S$:

- apply induction hypothesis for S m
- extend the function
- prove that bijection is preserved

$m \notin S$: use induction hypothesis

Overview of finite sets



Inclusion-exclusion principle

$$size(X \cup Y) + size(X \cap Y) = size(X) + size(Y)$$
$$= size(X \cup Y) + size(X \cap Y) = size(X \cap Y \cup X \setminus Y) + size(Y)$$
$$= size(X \cup Y) + size(X \cap Y) = size(X \cap Y) + size(X \setminus Y) + size(Y)$$
$$= size(X \cup Y) + size(X \cap Y) = size(X \cap Y) + size(X \setminus Y \cup Y)$$
$$= size(X \cup Y) + size(X \cap Y) = size(X \cap Y) + size(X \cup Y)$$

Prop vs bool

```
def mem: A -> K A -> bool
| x empty := false
| x {y} := x=y
| x (y ∪ z) :=
    (mem x z)= tt ∨
    (mem x y) = tt
```

```
def mem_prop: A -> K A ->
    Prop
| x empty := ff
| x {y} := x=y
| x (y ∪ z) :=
    (mem x z) ∨
    (mem x y)
```

Solution

```
lemma mem_iff_mem_prop (a:A) (X: K A):
  (mem a X = tt) ↔ mem_prop a X
```

Coercion

Function from one type into another

```
def coe_finset_set (S: listSet A) : set A
```

Efficient union

Multiple ways to compute $X \cup Y$

- Insert every element from X into Y
- Insert every element from Y into X
- Merge as lists and transform back to original type

Depending on the size of the lists different options are better.
Implementations in Coq select one for better runtime.

Difference II

```
def difference': binaryTree A -> binaryTree A ->
    binaryTree A
| t (binaryTree.empty) := t
| t (binaryTree.node tl x tr) := difference' (
    difference' (delete x t) tl) tr
```