Formalizing Combinatorics Definitions in the Lean Theorem Prover

Alena Gusakov

University of Waterloo

CanaDAM

Outline

- Formalizing mathematics: What is it and why do we care?
- The Lean algebra hierarchy: A template for formalizing theories?
- The Lean graph theory library: Why the algebra library design doesn't work

Formalizing Mathematics

Formalizing a mathematical theory is the process of expressing it precisely in a logical framework, usually in a proof assistant.

We do it for many reasons, including searching for mistakes, e.g.

- Terry Tao led a team in formalizing the proof of the PFR Conjecture, and found a mistake in one of the lemmas in the proof! (This was corrected in the formalization).

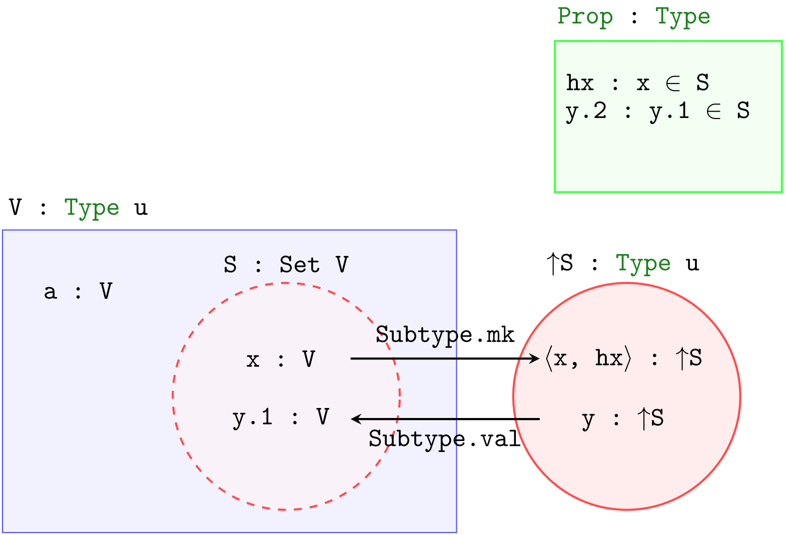
Formalizing also leads us to new and interesting insights about the theory we formalize.

Dependent Type Theory

Dependent type theory is a grammar for expressing mathematical statements.

Everything is a term or a type, and every term has a *unique* type.

Types, Subtypes, Sets



Simple graphs in mathlib

Question: Do we want to treat the vertices of a graph as a type, or a set?

mathlib's Algebra Hierarchy

Algebraic objects, i.e. groups, rings, semigroups, etc are `class`es defined on a `Type`.

- We can easily express statements without type coercions.
- We can take advantage of "ad hoc polymorphism," i.e. the dependent type theory equivalent of inheritance of properties.

Algebraic subobjects, i.e. subgroups, subrings, submodules, etc are `class`es defined on `Set`s.

- We can avoid type coercions for the most part, because the binary operation is still defined on the `Type` and not the `Set`.

Groups

This is a simplified version of the definition of a group, we have a lot of syntactic sugar and additional definitions that go into this.

```
class Group (G : Type u) extends DivInvMonoid G where
  /-- Binary operation denoted by `*` -/
  binary_op : α → α → α
  /-- Associativity -/
  op_assoc : ∀ a b c : G, a * b * c = a * (b * c)
  /-- Identity element, denoted by `1` -/
  one : α
  one_mul : ∀ a : M, 1 * a = a
  /-- Invert an element of α, denoted by `a⁻¹`. -/
  inv : α → α
  inv_mul_cancel : ∀ a : G, a⁻¹ * a = 1
```

Subgroups

```
structure Subgroup (G : Type*) [Group G] where
  carrier : Set G
  mul_mem {a b} : a ∈ carrier → b ∈ carrier → a * b ∈
    carrier
  /-- A subgroup contains 1‘. -/
  one_mem : (1 : M) ∈ carrier
  /-- ‘s‘ is closed under inverses -/
  inv_mem : ∀ {s : S} {x}, x ∈ s → x⁻¹ ∈ s
```

Subgroups vs Subgraphs

Similarities: Both are structures defined on subsets of the parent object, inheriting certain properties and requiring their own versions, e.g.

- A subgroup inherits the binary operation, and must be closed under the binary operation.
- A subgraph is a subset of the binary relation, which must still be symmetric and inherits irreflexivity.

Subgroups vs Subgraphs

Differences:

- Hierarchies:
  - Algebraic objects have an inheritance hierarchy, where e.g. the definition of a ring builds on the definition of a group.
  - Graphs don't have a hierarchy - e.g. we can think of an undirected graph as a digraph where we forget edge orientations, or we can think of a digraph as an undirected graph where we choose edge orientations.
- What we do with groups is often very different from what we do with graphs, more on this later...

Simple graphs in mathlib

The simple graph hierarchy was designed to imitate that of the algebra library.

- `SimpleGraph` is defined as a symmetric, irreflexive adjacency relation on a vertex type `V`.
- If `G : SimpleGraph V`, then `Subgraph G` is defined* as an adjacency relation on vertex type `Set V`.

*We also have an `IsSubgroup` predicate for `SimpleGraphs`.

Simple graphs in mathlib

```
structure SimpleGraph (V : Type u) where
  /-- The adjacency relation of a simple graph. -/
  Adj : V → V → Prop
  symm : Symmetric Adj
  loopless : Irreflexive Adj
```

```
structure Subgraph {V : Type u} (G : SimpleGraph V) where
  /-- Vertices of the subgraph -/
  verts : Set V
  /-- Edges of the subgraph -/
  Adj : V → V → Prop
  adj_sub : ∀ {v w : V}, Adj v w → G.Adj v w
  edge_vert : ∀ {v w : V}, Adj v w → v ∈ verts
  symm : Symmetric Adj := by aesop_graph
```

Simple graphs in mathlib

The `SimpleGraph` definition has some nice properties, and in some ways is easy to work with.

However, there is a significant drawback: it is difficult to work with common graph operations, substructures, etc!!

Vertex Deletion

Vertex deletion (or, equivalently, graph restriction) can be defined on our definition of `SimpleGraph V` in two ways:

1. Output `SimpleGraph W`, where `W` is a subtype derived from `V` by deleting everything in `S`, e.g. `SimpleGraph (univ \ S)` where `univ \ S` is coerced to a type
2. Output `Subgraph G`, where `verts` is the set complement of `S` and has type `Set V`.

In the second case, we still have to perform type coercions: Any time we want to use `SimpleGraph` lemmas on a `Subgraph`, we have to use a type coercion...or copy all the lemmas for `Subgraph`. This is **bad practice**.

Walks in simple graphs

We define walks in simple graphs inductively, i.e.

```
inductive Walk : V → V → Type u
  | nil {u : V} : Walk u u
  | cons {u v w : V} (h : G.Adj u v) (p : Walk v w) :
    Walk u w
```

In this definition, `nil` is the type of empty walks, and `cons` requires us to provide a proof that `u` is adjacent to `v` in order for us to append edge `G.adj u v` to `Walk v w`.

Walks in subgraphs

If we want to define walks in a subgraph, we have two options:

1. Coerce `Subgraphs` to `SimpleGraphs` every time we want to use `Walk`
2. Make a new `Walk` definition for `Subgraph`, along with new lemmas and additional definitions

Once again, both options introduce a lot of extra work.

Graph Operations

Regardless of what we do, we end up having to deal with either a lot of type coercions or a lot of code duplication.

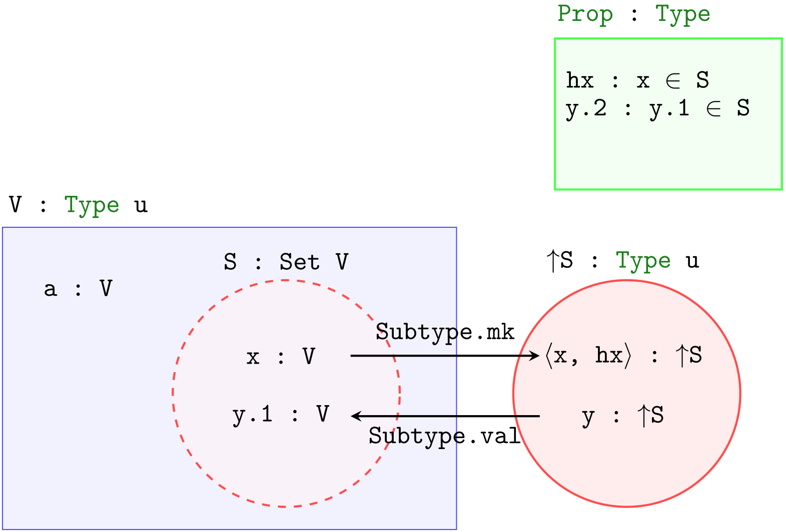
This might indicate that our definitions could be better...

Groups vs Graphs, Revisited

The key observation here is that in graph theory, every possible subobject is a graph, i.e. any combination of a subset of the vertex set and subset of the edge set will still give us some kind of graph.

On the other hand, deleting an element of a group gives us...something that is not usually a group.

Recall: Types, Subtypes, Sets



New Definitions

We are led to the conclusion that it is better to work with `Sets` in ambient `Types`, as opposed to imitating the algebra hierarchy with objects on `Types` and subobjects on `Sets`.

There are new PRs to mathlib's combinatorics library, with the definition of multigraphs implemented and accepted.

Multigraphs on Sets

```
structure Graph (α β : Type*) where
  /-- The vertex set. -/
  vertexSet : Set α
  /-- If `G.IsLink e x y` then we refer to `e` as `edge`
    and `x` and `y` as `left` and `right`. -/
  IsLink : β → α → α → Prop
  /-- The edge set. -/
  edgeSet : Set β := {e | ∃ x y, IsLink e x y}
  isLink_symm : ∀ ⦃e⦄, e ∈ edgeSet →
    (Symmetric <| IsLink e)
  eq_or_eq_of_isLink_of_isLink : ∀ ⦃e x y v w⦄, IsLink e
    x y → IsLink e v w → x = v ∨ x = w
  edge_mem_iff_exists_isLink : ∀ e, e ∈ edgeSet ↔ ∃ x y,
    IsLink e x y := by exact fun _ ↦ Iff.rfl
  left_mem_of_isLink : ∀ ⦃e x y⦄, IsLink e x y → x ∈
    vertexSet
```