How (Not) to Prove Theorems About Algorithms

(or; *fun with inductive types!*)

Jack Crawford

MATH3349: Special Topics in Mathematics

Automated and Interactive Theorem Proving

November 16, 2018

Overview

Automated Theorem Proving

Curry-Howard-Lambek Correspondence:

- *Proofs as Programs*
- *Propositions as Types*

Figure: Haskell Curry

Figure: Joachim Lambek

Automated Theorem Proving

By "proving" we usually just mean proof verification.

An automated theorem prover won't necessarily do any of the work for us.

Interactive Theorem Proving

Tools to help us understand and write our proofs

Does a bit of the grunt work for us, makes writing proofs feel more natural

```
1  theorem and_commutative {p q : Prop} : p ∧ q → q ∧ p :=
2  begin
3      intro h,
4      cases h with hp hq,
5      constructor,
6      repeat {assumption},
7  end
```

Tactic State — Updating

```
p q : Prop,
h : p ∧ q
⊢ q ∧ p
```

```
1  theorem and_commutative {p q : Prop} : p ∧ q → q ∧ p :=
2  begin
3      intro h,
4      cases h with hp hq,
5      constructor,
6      repeat {assumption},
7  end
```

Tactic State — Updating

```
p q : Prop,
hp : p,
hq : q
⊢ q ∧ p
```

```
1  theorem and_commutative {p q : Prop} : p ∧ q → q ∧ p :=
2  begin
3      intro h,
4      cases h with hp hq,
5      constructor,
6      repeat {assumption},
7  end
```

Tactic State — Updating

```
2 goals
p q : Prop,
hp : p,
hq : q
⊢ q

p q : Prop,
hp : p,
hq : q
⊢ p
```

What is Lean?

- First launched by Microsoft Research in 2013
- Current version is Lean 3
- Mathematics component library (*'mathlib'*) developed primarily at Carnegie Mellon (CMU).
- Metaprogramming of tactics occurs within Lean itself
- Dependently typed (with Sigma- and Pi-types you might be familiar with from Coq)
- Equipped with Calculus of Inductive Constructions (CIC)

Calculus of Inductive Constructions (CIC)

An inductive type consists of a name and a list of constructors.

A surprising amount of mathematical (or computational) objects can be defined using only inductive types.

```
inductive or (p q : Prop) : Prop
| mk_left  : p → or
| mk_right : q → or

example {p q : Prop} : or p q → or q p:=
begin
    intro h,
    cases h with hp hq,
    from or.mk_right _ hp,
    from or.mk_left _ hq,
end
```

```
Tactic State

2 goals
case or.mk_left
p q : Prop,
hp : p
⊢ or q p

case or.mk_right
p q : Prop,
hq : q
⊢ or q p
```

Figure: Logical 'or' defined inductively

```
inductive binary_tree (α : Type)
| leaf : α → binary_tree
| node : α → binary_tree → binary_tree → binary_tree

def sum_tree {α : Type} [field α] : (binary_tree α) → α
| (binary_tree.leaf a) := a
| (binary_tree.node a t₁ t₂) := a + (sum_tree t₁) + (sum_tree t₂)
```

Figure: Binary tree defined inductively

Calculus of Inductive Constructions (CIC)

As I come to discover, a clever use of inductive types is incredibly helpful (if not essential) for proving theorems about algorithms.

Still a lot of choice in how exactly we implement them, though, with non-trivial consequences.

Let's build something.

Spent most of Term 2 working on an implementation of Gaussian Elimination for the math library.

Let's build something.

~~Spent most of Term 2 working on an implementation of Gaussian Elimination for the math library.~~

OK, spent very little time implementing Gaussian Elimination, but spent most of Term 2 trying to prove anything at all about it.

Where to start?

Row Equivalence, of course.

What does row equivalence between M and N look like?

- A list of row operations (matrices)
- Multiplying all of these row operations in succession by M should yield N.
- Each row operation either:
  - scales a row;
  - swaps two rows; or,
  - adds a linear multiple of one row to another.

A first attempt

This checks all the boxes, what could go wrong?

```
44  variables {m n : ℕ} {α : Type} [field α]
45
46  def is_scale (M : matrix (fin m) (fin m) α) : Prop := ∃ (i₁ : fin m) (s : α) (hs : s ≠ 0), M = scale i₁ s hs
47  def is_swap (M : matrix (fin m) (fin m) α) : Prop := ∃ (i₁ i₂ : fin m), M = swap i₁ i₂
48  def is_linear_add (M : matrix (fin m) (fin m) α) : Prop := ∃ (i₁ i₂ : fin m) (s : α), M = linear_add i₁ s i₂
49
50  def is_row_operation (M : matrix (fin m) (fin m) α) := is_scale M ∨ is_swap M ∨ is_linear_add M
51
52  class row_equivalence (M N : (matrix (fin m) (fin n) α)) :=
53  (steps : list (matrix (fin m) (fin m) α))
54  (steps_implement : list.foldr (matrix.mul) M steps = N)
55  (steps_row_operations : ∀ (k : fin steps.length), is_row_operation (list.nth_le steps k k.is_lt))
```

Figure: I actually lost the code to my very first iteration, so this is a rough recreation. I think this is actually somehow slightly better than the original.

```
def example_algorithm (M : matrix (fin m) (fin n) α) (i₁ i₂ : fin m) : (matrix (fin m) (fin n) α) :=
begin
    from (swap i₁ i₂).mul M
end
```

It should be pretty easy to prove this is row equivalent, right?

Row Equivalence

```
def example_algorithm (M : matrix (fin m) (fin n) α) (i₁ i₂ : fin m) : (matrix (fin m) (fin n) α) :=
begin
    from (swap i₁ i₂).mul M
end
```

It should be pretty easy to prove this is row equivalent, right?

```
example {M : matrix (fin m) (fin n) α} {i₁ i₂ : fin m} : row_equivalence M (example_algorithm M i₁ i₂) :=
begin
    constructor,
    show list _,
    from list.cons (swap i₁ i₂) list.nil, -- Provide the list. Easy.
    refl, -- Prove that folding the list achieves the right result. Easy.
    intros,
    apply or.inr,
    apply or.inl, -- Unfolding is_row_operation to get is_swap as the goal
    constructor,
    constructor,
    show fin m, from i₂,
    show fin m, from i₁, --Instantiate the ∃ in is_swap with our indices
    simp,
    have k_eq_zero : k.val = 0, -- I have to prove that < 1 → k.val = 0
    cases k,                    -- just so I can prove that the kth elmt
    simp,                       -- in the list is also the zeroth elmt
    cases k_val,                -- so that I can simplify the statement
    simp,           -- ⊢ list.nth_le [swap i₁ i₂] (k.val) _ = swap i₁ i₂
    exfalso,                  -- to ⊢ swap i₁ i₂ = swap i₁ i₂
    from nat.not_lt_zero k_val (nat.lt_of_succ_lt_succ k_is_lt),
    have H₁ : list.nth_le [swap i₁ i₂] (k.val) _ = list.nth_le [swap i₁ i₂] (0) _,
    congr,
    assumption,
    rw H₁,
    simp, -- There we go
    simp,
    from nat.zero_lt_succ 0 -- and a proof that 0 < 1, for some reason.
end
```

Wrong.

What went wrong?

Recall from earlier, we thought:
What does row equivalence between M and N look like?

- A list of row operations (matrices)

Because row equivalence is 'list-like', we tried implementing it with a list.

What went wrong?

Recall from earlier, we thought:
What does row equivalence between M and N look like?

- A list of row operations (matrices)

Because row equivalence is 'list-like', we tried implementing it with a list.

Key observation: Don't implement 'list-like' things with a list.
Implement them 'like' a list: with an inductive type!

A (slightly) better use of inductive types

Define a single row equivalence step as an inductive type, and a full row equivalence by chaining steps together.

```
inductive row_equivalent_step : matrix (fin m) (fin n) α → matrix (fin m) (fin n) α → Type
| scale : Π (M : matrix (fin m) (fin n) α) (i₁ : fin m) (s : α) (hs : s ≠ 0),
    row_equivalent_step M (scaled M i₁ s hs)
| swap : Π (M : matrix (fin m) (fin n) α) (i₁ i₂ : fin m),
    row_equivalent_step M (swapped M i₁ i₂)
| linear_add : Π (M : matrix (fin m) (fin n) α) (i₁ : fin m) (s : α) (i₂ : fin m) (h : i₁ ≠ i₂),
    row_equivalent_step M (linear_added M i₁ s i₂)

def row_equivalent_step.elementary :
    Π {M N :  matrix (fin m) (fin n) α}, row_equivalent_step M N → matrix (fin m) (fin m) α
| M _ (row_equivalent_step.scale _ i₁ s h) := scale i₁ s h
| M _ (row_equivalent_step.swap _ i₁ i₂) := swap i₁ i₂
| M _ (row_equivalent_step.linear_add _ i₁ s i₂ h) := linear_add i₁ s i₂ h

inductive row_equivalent : matrix (fin m) (fin n) α → matrix (fin m) (fin n) α → Type
| nil : Π {N M : matrix (fin m) (fin n) α} (h : row_equivalent_step N M), row_equivalent N M
| cons : Π {N M L : matrix (fin m) (fin n) α} (h₁ : row_equivalent N M) (h₂ : row_equivalent_step M L), row_equivalent N L
```

Figure: This code has also been pretty heavily adapted for the presentation and looks a lot cleaner than it originally did. The functions *scale*, *swap*, and *linear_add* did not exist and I had implemented them explicitly in *elementary*.

We now require the fact that multiplication by an elementary matrix is equivalent to applying the row operation that the elementary matrix comes from. This is OK, because we were going to have to show this eventually, anyway.

```
def elementary_implements : -- This is actually not trivial, but let's assume we've shown it.
  Π {M N :  matrix (fin m) (fin n) α} (h : row_equivalent_step M N),  matrix.mul (row_equivalent_step.elementary h) M = N
    := sorry

example {M : matrix (fin m) (fin n) α} {i₁ i₂ : fin m} : row_equivalent M (example_algorithm M i₁ i₂) :=
begin
    constructor, -- Construct a row_equivalent from a row_equivalent_step
    dsimp[example_algorithm], -- Unfolds the algorithm as a swap statement
    have H₁, from elementary_implements (row_equivalent_step.swap M i₁ i₂),
    dsimp[row_equivalent_step.elementary] at H₁,
    rw H₁, -- Rewrite the (M.mul swap i₁ i₂) in the goal as (swapped M i₁ i₂)
    from row_equivalent_step.swap M i₁ i₂, -- yields row_equivalent M (swapped M i₁ i₂)
end
```

The rest of the proof is little bit easier this time, but still not ideal. In particular, invoking *elementary_implements* is a bit annoying.

Re-write the algorithm in terms of *row_reduction_steps*

This cuts the proof in half, but now makes our 'algorithm' more complicated than it needs to be.

```
def example_algorithm2 (M : matrix (fin m) (fin n) α) (i₁ i₂ : fin m) : (matrix (fin m) (fin n) α) :=
(row_equivalent_step.swap M i₁ i₂).elementary.mul M

example {M : matrix (fin m) (fin n) α} {i₁ i₂ : fin m} : row_equivalent M (example_algorithm2 M i₁ i₂) :=
begin
    constructor, -- Construct a row_equivalent from a row_equivalent_step
    dsimp[example_algorithm2], -- Unfolds the algorithm as a swap statement
    rw elementary_implements, -- We can now jump straight to a rw
    apply row_equivalent_step.swap, -- ⊢ row_equivalent_step M (swapped M i₁ i₂)
end
```

Shouldn't need to construct a *row_equivalent_step* first if we just want an elementary matrix. How do we improve this?

Final implementation of row equivalence

Boil down the 'essence' of a row operation in a neutral way with *elementary*.

```
inductive elementary (m : ℕ)
| scale : Π (i₁ : fin m) (s : α) (hs : s ≠ 0), elementary
| swap : Π (i₁ i₂ : fin m), elementary
| linear_add : Π (i₁ : fin m) (s : α) (i₂ : fin m) (h : i₁ ≠ i₂), elementary

variable {α}
def elementary.to_matrix {m : ℕ} : elementary α m → matrix (fin m) (fin m) α
| (elementary.scale i₁ s hs) := λ i j, if (i = j) then (if (i = i₁) then s else 1) else 0
| (elementary.swap _ i₁ i₂) :=  λ i j, if (i = i₁) then (if i₂ = j then 1 else 0) else if (i = i₂) then (if i
| (elementary.linear_add i₁ s i₂ h) := λ i j, if (i = j) then 1 else if (i = i₁) then if (j = i₂) then s else

def elementary.apply :  elementary α m →  (matrix (fin m) (fin n) α) →  matrix (fin m) (fin n) α
| (elementary.scale i₁ s hs) M := λ i j, if (i = i₁) then s * M i j else M i j
| (elementary.swap _ i₁ i₂) M := λ i j, if (i = i₁) then M i₂ j else if (i = i₂) then M i₁ j else M i j
| (elementary.linear_add i₁ s i₂ h) M := λ i j, if (i = i₁) then M i j + s * M i₂ j else M i j

structure row_equivalent_step (M N : matrix (fin m) (fin n) α) :=
(elem : elementary α m)
(implements : matrix.mul (elem.to_matrix) M = N)
```

Any simple 'algorithm' as from earlier can now be proved just using *...of_elementary* or *...of_elementary_apply*.

```
lemma elementary.mul_eq_apply :
  Π {M : matrix (fin m) (fin n) α} (e : elementary α m), ((e.to_matrix).mul M) = (e.apply M) :=
    sorry -- We still haven't handled this yet. It will come.

def row_equivalent_step.of_elementary :
  Π {M : matrix (fin m) (fin n) α} (e : elementary α m), row_equivalent_step M ((e.to_matrix).mul M) :=
    λ _ _, ⟨_, by refl⟩

def row_equivalent_step.of_elementary_apply :
  Π {M : matrix (fin m) (fin n) α} (e : elementary α m), row_equivalent_step M (e.apply M) :=
begin
  intros,
  rw ←elementary.mul_eq_apply,
  apply row_equivalent_step.of_elementary
end
```

Interlude: How do we prove multiplication by elementary matrix is equal to 'applying' the row operation, anyway?

Interlude: How do we prove multiplication by elementary matrix is equal to 'applying' the row operation, anyway?

It took about 15 lemmas.

These were tedious, but relatively straightforward:

```
@[simp] lemma mul_scale_scaled {i : fin m} {j : fin n} {s : α} {h : s ≠ 0} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.scale i s h).to_matrix M) i j = s * M i j := sorry

@[simp] lemma mul_swap_swapped_1 {i₁ i₂ : fin m} {j : fin n} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.swap _ i₁ i₂).to_matrix M) i₁ j = M i₂ j := sorry

@[simp] lemma mul_swap_swapped_2 {i₁ i₂ : fin m} {j : fin n} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.swap _ i₁ i₂).to_matrix M) i₂ j = M i₁ j := sorry

@[simp] lemma mul_linear_add_added {i₁ i₂ : fin m} {s : α} {j : fin n} {h : i₁ ≠ i₂} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.linear_add i₁ s i₂ h).to_matrix M) i₁ j = M i₁ j + s * M i₂ j := sorry
```

Interlude: How do we prove multiplication by elementary matrix is equal to 'applying' the row operation, anyway?

It took about 15 lemmas.
These were tedious, but relatively straightforward:

```
@[simp] lemma mul_scale_scaled {i : fin m} {j : fin n} {s : α} {h : s ≠ 0} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.scale i s h).to_matrix M) i j = s * M i j := sorry

@[simp] lemma mul_swap_swapped_1 {i₁ i₂ : fin m} {j : fin n} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.swap _ i₁ i₂).to_matrix M) i₁ j = M i₂ j := sorry

@[simp] lemma mul_swap_swapped_2 {i₁ i₂ : fin m} {j : fin n} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.swap _ i₁ i₂).to_matrix M) i₂ j = M i₁ j := sorry

@[simp] lemma mul_linear_add_added {i₁ i₂ : fin m} {s : α} {j : fin n} {h : i₁ ≠ i₂} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.linear_add i₁ s i₂ h).to_matrix M) i₁ j = M i₁ j + s * M i₂ j := sorry
```

Unfortunately, they required a couple deceptively simple-looking lemmas that took an adventure of their own to solve.

```
@[simp] lemma mul_scale_scaled {i : fin m} {j : fin n} {s : α} {h : s ≠ 0} {M : matrix (fin m) (fin n) α} :
  (matrix.mul (elementary.scale i s h).to_matrix M) i j = s * M i j :=
begin
  dsimp [matrix.mul],
  dsimp [elementary.to_matrix],
  simp, -- ⊢ finset.sum finset.univ (λ (x : fin m), ite (i = x) (s * M x j) 0) = s * M i j
  rw finset.sum_ite_zero,
end
```

```
lemma finset.sum_ite_zero
  {α : Type*} [fintype α] [decidable_eq α] (a₀ : α) {β : Type*} [add_comm_monoid β] [decidable_eq β] (f : α → β) :
  finset.sum finset.univ (λ a, ite (a₀ = a) (f a) 0) = f a₀ := sorry
```

Figure: In case you forgot just how much more tedious automated theorem proving can be than just convincing a human.

The closest thing to this statement in *mathlib* was the statement that:

- The sum of a single finitely-supported function over its (singleton) support is the function evaluated at the point.

Not much to work with.

Had to prove:

1. There is a function which is finitely-supported over a singleton set which does the same thing as the *ite*.
2. Hence, this is a single finitely-supported function.
3. The sum by a finitely-supported function over a set which contains its support is equal to summing the the same function over its support.
4. Restate *finset.sum* as *finsupp.sum*
5. The sum of a single finitely-supported function over its (single-point) support is the function evaluated at the point.

```
lemma finset.sum_ite_zero
  {α : Type*} [fintype α] [decidable_eq α] (a₀ : α) {β : Type*} [add_comm_monoid β] [decidable_eq β] (f : α → β) :
  finset.sum finset.univ (λ a, ite (a₀ = a) (f a) 0) = f a₀ := begin
    rw finsupp_of_ite_eq_ite, -- ⊢ finset.sum finset.univ ((finsupp_of_ite a₀ (λ (a : α), f a)).to_fun) = f a₀
    rw finsupp_of_ite_single, -- ⊢ finset.sum finset.univ ((finsupp.single a₀ (f a₀)).to_fun) = f a₀
    apply eq.trans (@finsupp_sum_support_subset α _ _ β _ (finsupp.single a₀ (f a₀)) finset.univ (by apply finset.subset_univ)),
    -- ⊢ finsupp.sum (finsupp.single a₀ (f a₀)) (λ (x : α) (y : β), y) = f a₀
    apply eq.trans (@finsupp.sum_single_index α β β _ _ _ _ a₀ (f a₀) (λ x y, y) (by refl)),
    -- ⊢ (λ (x : α) (y : β), y) a₀ (f a₀) = f a₀
    simp,
  end
```

This one was much worse.

```
lemma finset.sum_ite_zero₂
  {α : Type*} [fintype α] [decidable_eq α] (a₀ a₁ : α) {β : Type*} [add_comm_monoid β] [decidable_eq β] (f g : α → β) (h_ne : a₀ ≠ a₁):
  finset.sum finset.univ (λ a, ite (a₀ = a) (f a) (ite (a₁ = a) (g a) 0)) = f a₀ + g a₁ := sorry
```

Another bunch of (much larger) lemmas later, we eventually arrive at our destination.

An unfortunate reminder that automated theorem provers perhaps aren't quite ready for a lot of practical applications, yet.

End of detour: Back to Gaussian Elimination

Let's refresh – how does the algorithm go again?

1. Look down the column until we find a nonzero item and:
   i. move it to the top, or;
   ii. repeat the algorithm on the submatrix given by excluding the first column, if we can't find one.
2. Divide the pivot row by the value of the pivot, making it 1.
3. Iterate down the column from the pivot, subtracting multiples of the pivot row to set each value to zero.

How do we implement this in Lean?

We have two choices. We could either:

- implement a function that performs row reduction around just the first column, calls itself on the submatrix, and then combines them all together somehow; or,
- perform the algorithm 'in-place', never actually breaking the matrix up into submatrices, and instead just doing recursion over the location of the pivot.

The latter seemed to be a bit faster, and honestly, a bit easier.

For well-foundedness, we want to have a natural number which strictly decreases in size on every recursion of the algorithm.

What's the best candidate for this?
The number of columns to the right of (and including) the pivot.

We consider the position of the pivot relative to the bottom-right corner of the matrix.

Don't want to have to subtract position from the size of the matrix every time we need to read an element, though.

We choose to implement steps 1) and 3) of the algorithm in terms of the actual row and column index in the matrix.

We still have well-foundedness, and now we only need to perform the subtraction once and pass it into those steps, rather than having to do it individually within the steps.

Slightly modify our algorithm

To solve the problems with well-foundedness described above, we tweak our algorithm as follows:

1. Look up the column until we hit the pivot. Swap the first non-zero element we see with the pivot and continue.
2. If the pivot element is nonzero, divide the pivot row by the value of the pivot.
3. If the pivot element is zero, call the algorithm again but with the pivot position from the right decremented by one. Otherwise, clear the column from the bottom up and then call the algorithm again with the pivot position from both the bottom and the right each decremented by one.

Gaussian Elimination

```
def ge_aux_findpivot :
  (fin m) → (fin m) → (fin n) → (matrix (fin m) (fin n) α) → (matrix (fin m) (fin n) α)
| ⟨0, h₁⟩        i₀ j₀ M := M
| ⟨(k + 1), h₁⟩ i₀ j₀ M :=
        if k ≥ i₀.val then
            if M ⟨k+1, h₁⟩ j₀ ≠ 0
                then matrix.mul (elementary.swap α ⟨k+1, h₁⟩ i₀).to_matrix M
            else ge_aux_findpivot ⟨k, nat.lt_of_succ_lt h₁⟩ i₀ j₀ M
        else M

def ge_aux_improvepivot :
  (fin m) → (fin n) → (matrix (fin m) (fin n) α) → (matrix (fin m) (fin n) α) :=
λ i₀ j₀ M, if h : M i₀ j₀ ≠ 0
    then matrix.mul (elementary.scale i₀ ((M i₀ j₀)⁻¹) (inv_ne_zero h)).to_matrix M else M

def ge_aux_eliminate :
  Π (k : fin m) (i : fin m) (j : fin n) (M : matrix (fin m) (fin n) α) (h : M i j = 1), (matrix (fin m) (fin n) α) :=
λ k i j M h, if h₀ : k≠i then matrix.mul (elementary.linear_add k (-(M k j)) i h₀).to_matrix M else M

def ge_aux_eliminatecolumn :
  Π (k : fin m) (i : fin m) (j : fin n) (M : matrix (fin m) (fin n) α), (matrix (fin m) (fin n) α)
| ⟨0,h₁⟩ i j M := M
| ⟨k+1, h₁⟩ i j M := begin
    from if h : k < i.val then M else begin
        apply ge_aux_eliminatecolumn ⟨k, nat.lt_of_succ_lt h₁⟩ i j _ ,
        have h_ne : fin.mk (k+1) h₁ ≠ i,
        intros h_eq, apply h, cases h_eq, simp[nat.lt_succ_self 0],
        from matrix.mul (elementary.linear_add ⟨k+1, h₁⟩ (-(M ⟨k+1,h₁⟩ j)) i h_ne).to_matrix M,
    end
end
```

Gaussian Elimination

```
def ge_aux_findpivot_row_equivalent :
  Π (i : fin m) (iₒ : fin m) (jₒ : fin n) (M : matrix (fin m) (fin n) α), row_equivalent M (ge_aux_findpivot i iₒ jₒ M)
| ⟨0, hₒ⟩   iₒ jₒ M :=
begin
    simp[ge_aux_findpivot],
    from row_equivalent.nil,
end
| ⟨k+1, hₒ⟩ iₒ jₒ M := begin
    unfold ge_aux_findpivot,
    split_ifs,
    from @row_equivalent_step.of_elementary m n α _ _ M (elementary.swap α ⟨k + 1, hₒ⟩ iₒ),
    apply ge_aux_findpivot_row_equivalent,
    from row_equivalent.nil,
end

def ge_aux_improvepivot_row_equivalent :
  Π (iₒ : fin m) (jₒ : fin n) (M : matrix (fin m) (fin n) α), row_equivalent M (ge_aux_improvepivot iₒ jₒ M)
| iₒ jₒ M :=
begin
    simp[ge_aux_improvepivot],
    split_ifs,
    from @row_equivalent_step.of_elementary m n α _ _ M (elementary.scale iₒ (M iₒ jₒ)⁻¹ (ge_aux_improvepivot._proof_1 iₒ jₒ M h)),
    from row_equivalent.nil
end

def ge_aux_eliminate_row_equivalent :
  Π (i : fin m) (iₒ : fin m) (jₒ : fin n) (M : matrix (fin m) (fin n) α) (h : M iₒ jₒ = 1), row_equivalent M (ge_aux_eliminate i iₒ jₒ M h)
| i iₒ jₒ M h :=
begin
    unfold ge_aux_eliminate,
    split_ifs,
    from @row_equivalent_step.of_elementary m n α _ _ M (elementary.linear_add i (-M i jₒ) iₒ h_1),
    from row_equivalent.nil,
end
```

Our nice inductive types are robust enough to handle all of these proofs with ease.

These proofs look a *lot* worse otherwise (I tried.)

```
def ge_aux_eliminatecolumn_row_equivalent :
  Π (i : fin m) (iₒ : fin m) (jₒ : fin n) (M : matrix (fin m) (fin n) α), row_equivalent M (ge_aux_eliminatecolumn i iₒ jₒ M)
| ⟨0, hₒ⟩ iₒ jₒ M :=
begin
    unfold ge_aux_eliminatecolumn,
    from row_equivalent.nil,
end
| ⟨k+1, hₒ⟩ iₒ jₒ M :=
begin
    unfold ge_aux_eliminatecolumn,
    split_ifs,
    from row_equivalent.nil,
    apply row_equivalent.trans,
    show matrix (fin m) (fin n) α,
    from (matrix.mul
          (elementary.to_matrix
             (elementary.linear_add ⟨k + 1, hₒ⟩ (-M ⟨k + 1, hₒ⟩ jₒ) iₒ
                (ge_aux_eliminatecolumn._main._pack._proof_1 k hₒ iₒ h)))
          M),
    from @row_equivalent_step.of_elementary m n α _ _ M (elementary.linear_add ⟨k + 1, hₒ⟩ (-M ⟨k + 1, hₒ⟩ jₒ) iₒ
             (ge_aux_eliminatecolumn._main._pack._proof_1 k hₒ iₒ h)),
    apply ge_aux_eliminatecolumn_row_equivalent,
end
```

...Probably easier to switch out of the presentation and look at the code directly at this point.

Bonus: Also proved that row equivalences are invertible over division rings.

Great! But now what?

It took all of that to finally prove that for any matrix, there is an invertible matrix that you can multiply by to perform the action of Gaussian elimination, which yields a result that is equal to 'applying' Gaussian elimination. Phew! But what about...

Great! But now what?

It took all of that to finally prove that for any matrix, there is an invertible matrix that you can multiply by to perform the action of Gaussian elimination, which yields a result that is equal to 'applying' Gaussian elimination. Phew! But what about...

- proofs about the rank of the matrix?

Great! But now what?

It took all of that to finally prove that for any matrix, there is an invertible matrix that you can multiply by to perform the action of Gaussian elimination, which yields a result that is equal to 'applying' Gaussian elimination. Phew! But what about...

- proofs about the rank of the matrix?
- extending to Gauss-Jordan?

Great! But now what?

It took all of that to finally prove that for any matrix, there is an invertible matrix that you can multiply by to perform the action of Gaussian elimination, which yields a result that is equal to 'applying' Gaussian elimination. Phew! But what about...

- proofs about the rank of the matrix?
- extending to Gauss-Jordan?
- proving that the result of Gaussian elimination is in row echelon form (???)

Great! But now what?

It took all of that to finally prove that for any matrix, there is an invertible matrix that you can multiply by to perform the action of Gaussian elimination, which yields a result that is equal to 'applying' Gaussian elimination. Phew! But what about...

- proofs about the rank of the matrix?
- extending to Gauss-Jordan?
- proving that the result of Gaussian elimination is in row echelon form (???)
  - or defining row echelon form at all (?!?!?)

I'm working on it.

I'm working on it.

It may very well require tearing up everything I've done and reimplimenting it all from scratch (again). Let's hope not.

This project is on GitHub:
https://github.com/jjcrawford/lean-gaussian-elimination
jack.crawford@anu.edu.au
u6409041

Attributions:
Photograph of Haskell Curry by Gleb Svechnikov, distributed under a CC BY-SA 4.0 license.
Photograph of Joachim Lambek by Andrej Bauer, distributed under a CC BY-SA 2.5 si license.