Theorem Proving with Inductive Types **Jack Crawford**
MATH3349: Special Topics in Mathematics Automated and Interactive Theorem Proving
Supervisor: Assoc Prof. Scott Morrison Date: 16/11/18

Abstract

We consider the usefulness of inductive types in the construction of proofs about recursive algorithms, using an implementation of Gaussian Elimination in Lean 3 as a case study.

About Lean 3 and the Calculus of Inductive Constructions

Lean is a dependently-typed, functional theorem proving language designed by Microsoft Research and initially launched in 2013. The current version is Lean 3. It is equipped with the Calculus of Inductive Constructions (CIC), and also features the capacity for rich metaprogramming from within the Lean language itself.

The Calculus of Inductive Constructions (CIC) is the specification by which data structures can be implemented in Lean, and allows for the creation of inductive types. An inductive type is defined by three attributes: a name, an arity, and a set of constructors. [2] Each constructor takes the form of a function whose return type is the inductive type being defined. The inductive type in question may also appear as the type signature of an argument of one of its own constructors, allowing for well-founded recursion over instances of the inductive type. Examples of such useful inductive types include finite lists, finite trees, and the natural numbers.

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

Figure 1: A finite binary tree implemented self-referentially as an inductive type in Lean.

1 Implementing Row Equivalence

It might not be immediately evident why it would be necessary to implement a definition for row equivalence before implementing the Gaussian elimination algorithm. After all, nothing in the algorithm itself relies on the fact that any two things are row equivalent. Why would it not be sufficient that we go ahead and implement the row reduction algorithm first and come back to worry about whether we can prove its row equivalence properties later?

The answer is that the only tool we really have for proving theorems about recursive algorithms is good, old-fashioned mathematical induction. In particular, if we ever want to show that the result of Gaussian elimination is row-equivalent to the matrix we started with, we're going to need to show that each step of the Gaussian elimination algorithm is row-equivalent to the one before it. This is not something that comes easily unless a bit of careful consideration has been put into designing each step of the algorithm with this induction already in mind. This is a principle that would seem to be good advice in general when approaching the implementation of recursive algorithms for theorem proving. In this case, it will require us to have a clearer understanding of exactly how we will be implementing row equivalence before proceeding.

1.1 Iteration 1

Let us recall what it means for two matrices to be row-equivalent. Suppose we have M and N, each an $m \times n$ matrix defined over the ring α. Then M and N are row-equivalent if:

- There exists a list of row operations (matrices), L;
- Multiplying each successive matrix in L in order T such that $TM = N$; and,
- Each matrix/row operation in L either:
  - scales a row i_1 by a non-zero scalar $s \in \alpha$;
  - swaps two rows i_1 and i_2; or,
  - adds a linear multiple (with factor $s \in \alpha$) of one row i_2 to another row i_1, where $i_1 \neq i_2$.

This would appear to be a relatively straightforward specification to implement in any dependently-typed language, although we shall see that there remains a great deal of subtlety in how the implementation is realised.

Consider, for example, the following implementation:

```
variables {m n : ℕ} {α : Type} [ring α]

def is_scale (M : matrix (fin m) (fin m) α) : Prop := ∃ (i₁ : fin m) (s : α) (
    hs : s ≠ 0), M = scale i₁ s hs
def is_swap (M : matrix (fin m) (fin m) α) : Prop := ∃ (i₁ i₂ : fin m), M =
    swap i₁ i₂
def is_linear_add (M : matrix (fin m) (fin m) α) : Prop := ∃ (i₁ i₂ : fin m) (
    s : α) (h : i₁ ≠i₂), M = linear_add i₁ s i₂ h

def is_row_operation (M : matrix (fin m) (fin m) α) := is_scale M ∨ is_swap M
    ∨ is_linear_add M

class row_equivalence (M N : (matrix (fin m) (fin n) α)) :=
(steps : list (matrix (fin m) (fin m) α))
(steps_implement : list.foldr (matrix.mul) M steps = N)
(steps_row_operations : ∀ (k : fin steps.length), is_row_operation (list.
    nth_le steps k.val (k.is_lt)))
```

where *scale*, *swap*, and *linear_add* (omitted for brevity) are functions which return the respective row-operation matrices.

This implementation would seem to fulfil all of the aforementioned properties of row equivalence, translated across quite faithfully into Lean. But do not be fooled - this is not what we would call a 'good' implementation. To demonstrate this, let us consider how well it performs in proving something trivial, such as the following 'algorithm':

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

Surely, if our definition of row equivalence were remotely sensible, it should take at most a couple lines to produce a proof that the output of this function is row-equivalent with its input.

Let us see how complicated it really is with our choice of definition for row equivalence we prepared above:

```
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 k < 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
```

This certainly isn't the most minimal form of the proof, but it is adequate for demonstrating a point: our implementation of row equivalence *sucks*.

So, where did we go wrong?
In the above proof, the first four lines satisfy two of the three attributes of our *row_equivalence* class immediately. The rest is dedicated to reading the first (and only) element out of the list of steps to verify that it does indeed satisfy the condition of being a row operation. It requires us to play games with the length of the list to prove that the index of the list that we're trying to read really is just 0. It would seem that just reading from the list is the action causing us so much trauma. It also appears as though this problem would increase linearly with the number of terms in our list, which makes this implementation completely unscalable. But if we recall from earlier, we recognised that an essential feature of row equivalence is that it essentially consists of a list of row operations satisfying certain conditions. How, then, might we attempt to implement such a 'list-like' datatype without the use of lists?

As we shall see, the solution is not to implement a 'list-like' datatype *using* a list, but to implement it *like* a list, which is to say, implementing it as an inductive type in the same way that lists are implemented.

1.2 Iteration 2

Taking into consideration everything we've learned from our first iteration, let us attempt to formalise row equivalence yet again, but this time leveraging the power of inductive types to our advantage.

Another detail that we also neglected to consider in our first implementation, but that we shall attempt to make use of now, is that there are two separate perspectives that can be taken towards row operations. Above, we spoke of row operations as elementary matrices that we might multiply by, with the effect that they perform some sort of intelligible manipulation on the rows of the matrix. Instead, we might talk about row operations purely in terms of the way they manipulate the rows of the matrix, with the fact that this is equivalent to multiplication by some elementary matrix as an afterthought. We do exactly this in the following implementation:

```
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
```

where *scaled*, *swapped*, and *linear_added* (omitted for brevity) are functions which return the result of 'applying' the respective row operations to a given matrix.

The way that we are chaining together our individual *row_equivalent_step*s with *row_equivalent* seems to be a far more scalable solution than our previous implementation with a list. We also have the fact that each step of the row equivalence really does just associate a matrix with the version of itself after application of a row operation definitionally built into our *row_equivalent_step* type, which would seem to be a sensible idea *(for now)*.

Then if we wanted to prove that the result of our example 'algorithm' from earlier (which is defined in terms of multiplication by an elementary matrix) is row-equivalent with its input, we would need to first prove that multiplication by an elementary matrix is equivalent to 'applying' the row operation to the matrix. Let's leave this *sorried* for now:

```
def elementary_implements : Π {M N : matrix (fin m) (fin n) α} (h :
    row_equivalent_step M N), (row_equivalent_step.elementary h).mul M = N :=
    sorry
```

We will be revisiting this in due time.

Now, let us consider how much simpler our proof of row equivalence becomes:

```
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 improvements are already apparent. This new proof is only about a third of the length of the old one. Despite this, it still would seem a bit more complicated than need be, given the triviality of the statement we're trying to demonstrate. Again, there are some further improvements to be made.

The part of the above proof that would appear to be the most annoying is the way that we are forced to invoke *elementary_implements*. Ideally, it should take at most one line to use the statement, but in this proof it is stretched out over three lines, and involves the unpleasant use of a *have* statement. The simplest way to resolve this is actually to re-write our 'algorithm' in terms of row equivalence steps to begin with, which would save us from the superfluous *dsimp* and *rw* above.

Consider that we could instead try the following:

```
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
```

This attains the desired result of cutting down the size of our proof by a few lines, but also makes implementing our example 'algorithm' significantly more complicated. If we look closely at *example_algorithm2*, an unfortunate bi-product of our implementation should become immediately obvious. Why do we need to pass M in to *row_equivalent_step.swap* just to take the elementary matrix of it again, essentially 'throwing away' the M that we just passed into it? This was a direct result of our choice to implement *row_equivalent_step* as our most fundamental representation of row operations, with row operation 'application' built in, and with multiplication by elementary matrices as an afterthought built on top.

It would seem that we might benefit from creating a way of representing row operations purely by their essentials (their parameters), with facts about how they are 'applied' to matrices, or what their elementary matrices might be, each implemented separately. But is there a way that we can do this, while still keeping the fact that row equivalence steps must represent some sort of relationship between one matrix and another given by a row operation? The answer is an enthusiastic *'absolutely!'*, and it might seem that we have an even cleaner way of doing it than before.

1.3 Iteration 3

Following on from all of the observations we have made from the previous two iterations, let us see if we can produce a third (and hopefully final) implementation that captures all of the behaviours described above in a more succinct and manageable way.

Consider the following implementation:

```
variables {m n : ℕ} (α : Type u) [ring α] [decidable_eq α]

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 i₁ = j then 1 else 0) else if (i = j) then 1
   else 0
| (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 0 else 0

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)

inductive row_equivalent : matrix (fin m) (fin n) α → matrix (fin m) (fin n)
    α → Type u
| nil : Π {N : matrix (fin m) (fin n) α}, row_equivalent N N
| 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
```

This implementation synthesises all of the insights we have gained so far into the best format yet. Note that the function lambdas above are replacing the functions we were using earlier (*swap*, *scale*, *linear_add*, *swapped*, *scaled*, *linear_added*). It turns out that defining these functions separately isn't actually really very useful for us (the only real reason we had defined them separately earlier was an aesthetic choice). An astute reader will recognise that there is still something quite important missing from above: the relationship between *elementary.apply* and *elementary.to_matrix*. We complete the core part of the implementation with the following:

```
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
```

Then, to address our little 'algorithm' from earlier, all we would need to do is implement it in terms of *elementary.to_matrix*, and we will be able to 'prove' it immediately with *row_equivalent_step.of_elementa*
This is certainly more elegant than all former iterations.

We have finally settled on a best-practice for implementing our row equivalences with inductive types, but looking back, it might seem as though we haven't actually really to *proved* anything, yet. This observation would be correct. So far, we have obscured the actual proofs by sleight of hand, and a single 'sorry' statement in *elementary.mul_eq_apply*. We shall now make a temporary departure from the land of inductive types to discuss what might actually be required to prove such a statement.

1.4 Proof that the 'application' of a row operation is equivalent to multiplication by its corresponding elementary matrix.

This section does not make heavy or interesting use of inductive types, and so is not really essential to the topic of the report, but is included anyway as a potential topic of interest to the reader.

The exact details of our approach are too intricate to explore in great detail here, but an abridged summary is as follows.

Matrix multiplication in *mathlib* is currently implemented in terms of *finset.sum*. Unfortunately, it would seem that the library is currently lacking in a wealth of lemmas supporting *finset.sum*, and so attempting to prove anything that depends on it can in general be quite inconvenient. In particular, the only non-trivial part of our implementation was in proving two lemmas:

```
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

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 a
    ₀ + g a₁ := sorry
```

These lemmas are quite similar, so it should be possible to create a general lemmas that proves an entire class of problems of this nature, but we considered this to be outside the scope of the project. The fine details of the proofs for these are available on the GitHub repository, [1] but we found it necessary to rely heavily on the *finsupp* class (finitely supported functions, from *mathlib*). An outline of the proof of*finset.sum_ite_zero* is as follows:

1. We define a function which is finitely-supported over the singleton set a_0, which when applied to any element performs the same action as the if-then-else.
2. Hence, this is a *single* finitely-supported function. (*finsupp.single* is a definition of its own in *mathlib*).
3. The sum by a finitely-supported function over a set which contains its support is equal to the sum of the same function over its support.
4. Some manipulation occurs to transform a *finset.sum* as a *finsupp.sum*.
5. The sum of a single finitely-supported function over its (singleton) support is the function evaluated at the point.

Needless to say, an expansion of the *matlib* library's collection of lemmas about *finset.sum* to cover trivial lemmas such as this would be invaluable.

2 Gaussian Elimination

The primary concern that needs to be addressed when implementing a recursive algorithm in a theorem proving language is well-foundedness. While intuitively we understand that any implementation of Gaussian elimination will be well-founded, it will require a bit more effort to convince Lean's typechecker that our algorithm really does terminate. When we define a function recursively using the pattern-matcher, Lean effectively takes the parameters of our function in the order they are listed and attempts to create a lexicographic ordering out of them. For natural numbers, or *fin* parameters, this means that we need to always be decrementing a term. Let's refresh ourselves on what the Gaussian elimination algorithm does (ignoring the base case for now):

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 ca n't find one.
2. Divide the pivot row by the value of the pivot, setting the pivot value to 1.
3. Iterate down the column from the pivot, subtracting multiples of the pivot row to s et each value to zero.

However we eventually choose to implement the algorithm, we will need to identify a parameter (or lexicographic pair of parameters) which decrements on each recursion. Reading through the algorithm, we notice that no matter which branch we follow, we always end up moving the pivot one unit to the right. It should be apparent from this that a good candidate for a well-founded parameter should be the number of *remaining columns* (i.e., the number of columns to the right of and including the pivot).

This causes us a few problems, though. If we ever want to read a value from the matrix, for example, we would have to perform the subtraction necessary to convert our coordinates back into the standard (measured from top-left) orientation. It would be nice to avoid performing this conversion wherever possible, as subtracting *fin*-types requires us to provide some annoying proofs that the resulting numbers are still strictly less than the size of the matrix.
A solution to this problem is to implement the individual steps of the row reduction in terms of the standard (top-left oriented) matrix coordinates, but use the bottom-right distance to the pivot as our coordinates in the main 'loop'. Then we need only perform the subtraction once and can pass it in as an argument to the functions which perform each of the respective steps.

This requires us to modify our algorithm slightly, in a few minor ways. We may now restate the algorithm loosely in the following terms (still, ignorning the base case for now):

1. Iterate up the pivot 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, apply the algorithm again but with the pivot position from the right decremented by one. Otherwise, iterate up the column and subtract the appropriate multiple from the pivot row to clear the cell, and then call the algorithm again with the pivot coordinates (from both the bottom and the right) each decremented by one.

This would seem to be the best implementation we have imagined so far.

The last thing that remains for us to do is to put all of the above discussion into practice, and see if we can prove the row equivalence of the algorithm.
As discussed above, we will separate each of the steps of the algorithms into different functions. Here's an example of what the first step of the algorithm might look like, and the proof that its result is row-equivalent to its input:

```
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_findpivot_row_equivalent :
  Π (i : fin m) (i₀ : fin m) (j₀ : fin n) (M : matrix (fin m) (fin n) α),
    row_equivalent M (ge_aux_findpivot i i₀ j₀ M)
| ⟨0, h₀⟩   i₀ j₀ M :=
begin
    simp[ge_aux_findpivot],
    from row_equivalent.nil,
end
| ⟨k+1, h₀⟩ i₀ j₀ M := begin
    unfold ge_aux_findpivot,
    split_ifs,
    from @row_equivalent_step.of_elementary m n α _ _ M (elementary.swap α ⟨k
    + 1, h₀⟩ i₀),
    apply ge_aux_findpivot_row_equivalent,
    from row_equivalent.nil,
end
```

We can see above that proving row equivalence over step 1 was relatively painless, thanks to the efforts that we made earlier (in section 1) to carefully implement our inductive types. We will not bother inspecting the other two cases here, but they are openly accessible on GitHub. [1]
The remainder of the implementation is largely trivial at this point, and similar to the example demonstrated above.

Our implementation of the Gaussian elimination algorithm is now functionally complete, and when applied to any instance of the matrix computationally returns results that we would expect. We have also proven that the algorithm is well-founded and hence guaranteed to terminate, and that for any matrix M, the action of Gaussian elimination on M is equivalent to multiplication by some invertible matrix.* In the limited scope of this project and report we have neglected to address a number of features, including: proving further statements about Gaussian elimination, such as the fact that the result is in row-echelon form, or that it yields

*In a file titled *row_equivalence_fields.lean* (accessible in the GitHub repository), we demonstrate invertibility.

us a simple calculation for the matrix rank; the exact implementation of matrices chosen (it is now the standard implementation in *mathlib*), and the subsequent implications for complexity and processing speed; alternate ways we might have proven well-foundedness that potentially do not require subtracted coordinates; or how the algorithm might be extended to perform full Gauss-Jordan elimination. In particular, the computational time of the algorithm is *slow*, and the definition of row-echelon form is a very critical omission. This project is certainly capable of being extended in many ways, and will continue to be in active development on GitHub into the future.

References

[1] Jack Crawford. Lean gaussian elimination (github repository). https://github.com/jjcrawford/lean-gaussian-elimination, 2018.

[2] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions. *All about Proofs, Proofs for All*, 55, 2015.