Journal Report 19
2/24/20-2/28/20
Rahul Mittal
Computer Systems Research Lab
Period 2, White

Daily Log

Tuesday February 25

Continue Coq tutorial, successfully prove that the empty list append to a list l is the same list l, successfully prove that a list appended to a nonempty list results in a nonempty list, finish Coq tutorial.

Thursday February 27

Moving now to Andrew Haven, read through a bit, get to subset sizes section, start thinking about how to prove Colorado 1 using this stuff, Theorem choose_equal reduces $\binom{n}{k} = \binom{n}{n-k}$ to $|T| = n$ -> choose_cardinal T k = choose_cardinal T ($|T| - k$) where I'm not even sure if $|T| - k$ is defined by Haven, hopefully it is, first step to proving this is probably to prove that the relation f between subsets and their complement is a bijective function, this requires to prove that f x y -> f x y' -> y = y', but subsets don't seem to have that normal kind of equality, what if I just be lazy and make that a "theorem" that I don't actually prove, saying that forall (a b), (a <-> b) -> a = b, ok, to get rid of subtraction can phrase as for all (j k : nat), $j + k = |T|$ -¿ choose_cardinal T j = choose_cardinal T k, now need to figure out how to prove that complement of a set w/ cardinality j has cardinality k, might be easier to prove specifically for sets of natural numbers (which is sufficient).

Sunday March 1

Start proof of $\binom{n}{k} = \binom{n}{n-k}$, trying to first define 'complement', once again sidestep subtraction by requiring a proof that $j + k = |T|$ as an argument, now write "axiom" that (s1 <-> s2) -> s1 = s2, now need to prove that complement is a bijective function, first proving that it's a function, finally finished proof of complement_is_function, now need to prove that complement is injective, basically same thing, done with that, now need to prove surjectivity and totality, I'll do totality first, to prove existence have to demonstrate that set defined by elements of T not in x has cardinality k (where $j + k = |T|$), should be able to do that by showing $|x| + |y| = |T|$, $|x| + |y|$ is defined as $|x + y|$ where $x + y$ is disjoint sum, need to create a bijection between $x + y$ and T, time to do that, defined disjoint_bij, proving that disjoint_bij is a function, checked something in Coq to make sure that case works the way i want it to, ah but i can't prove equality here because the proofs that they satisfy the predicate aren't necessarily equal, finished proving that disjoint_bij is a function, now doing injective, that was a lot easier, now doing total, unfortunately, can't really do this because A A isn't generally true in Coq, skipping that for now, doing surjectivity instead, surjectivity was easy as well.

Timeline

Date	Goal	Met
February 17	Be able to find all implications between statements in Colorado proofs, start work on combinatorics foundation in Coq	Finished implications for Colorado 1, 2, 3, not started Coq
February 24	Start work on combinatorics foundation in Coq	Read most of Coq tutorial
March 2	Finish Coq tutorial, go back to Haven's thesis on using Coq for combinatorics, express Colorado 1 in Coq	Yes, yes, started
March 9	Complete Coq combinatorics foundation (state/prove necessary foundational results not already shown by Haven)	
March 16	Write code (in Kotlin) to generate Coq output for basic components of a proof	

Reflection

The first part of this week was fairly straightforward, just finishing the Coq tutorial which included actually writing some simple proofs in Coq related to lists. The majority of my time, though, was spent reading Andrew Haven's thesis ("Automated Proof Checking in Introductory Discrete Mathematics Classes") and trying to prove $\binom{n}{k} = \binom{n}{n-k}$, which apparently is extremely nontrivial (in Coq for someone who barely knows Coq).

Just in reading his thesis, I saw a lot of Coq constructs that weren't in the tutorial, but that's to be expected, and later on when I needed to actually know how to interact with them, it wasn't hard to look them up and understand them.

The problems came when I started to try to write my proof. Haven included two different definitions of $\binom{n}{k}$, an algebraic one defined recursively as $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, and one defined in terms of what $\binom{n}{k}$ actually means - the amount of subsets of size k of a set of size n. He also included a proof of their equality. You can easily then prove $\binom{n}{k} = \binom{n}{n-k}$ from the recursive definition, but my goal is to translate the proof, not just the result, so I needed to prove $\binom{n}{k} = \binom{n}{n-k}$ based on the latter definition.

Here's Colorado 1 again for reference:

We will show that both sides of the equation count the number of ways to choose a subset of size k from a set of size n. The left hand side of the equation counts this by definition. Now we consider the right hand side. To choose a subset of size k, we can instead choose the $n-k$ elements to exclude from the subset. There are $\binom{n}{n-k}$ ways to do this. Therefore the right hand side also counts the desired quantity.

And here's a proof that's more one-to-one with my intended proof in Coq:

For all integers n, k, for any set S of size n there exists a bijection between subsets of S of size k and subsets of S of size $n-k$, that being the relation between a subset and its complement. Therefore, the amount of subsets of S of size k is equal to the amount of subsets of S of size $n-k$, and so $\binom{n}{k} = \binom{n}{n-k}$.

The meaty part of the proof lies in proving that the relation between a subset and its complement its bijective, even though that's fairly obvious to us humans. I started by defining this relation:

```
Definition complement (T : Type) (j k : cardinality) (proof: j + k = |T|) (s1: size
    forall a : T, (s1 a \/ s2 a) /\ ~(s1 a /\ s2 a).
```

Now that I had the relation, in order to prove it bijective using Haven's framework, I needed to prove that it was a function, injective, surjective, and total. Because of the obvious symmetry in the relation, proving injectivity and surjectivity is identical to proving functionality (?) and totality respectively, so I only had two proofs to worry about.

Starting with proving that it was a function, aka that if x and y were related and x and y' were related then $y = y'$, I ran into the issue that these subsets (x, y, y' are subsets) are actually Props, or propositions, and since equality in Coq is based on equality of construction, the notion Coq had of equality between these props wouldn't allow me to prove equality, which is a problem considering that I needed to prove $y = y'$. I sidestepped this by adding as an axiom that $y \longleftrightarrow y' \implies y = y'$. This seemed hacky and unideal when I was originally doing it, but it's apparently a common thing, and it makes sense logically, so this isn't actually much of a problem. I made sure to only axiomatize this specifically for subsets as defined by Haven, so it wouldn't apply generally.

Here's my "axiom" and the proof that complement is a function:

```
Lemma subset_eq (T : Type) (k : cardinality) (s1 s2 : sized_subset T k) :
        (forall a : T, s1 a <-> s2 a) -> s1 = s2
    admit.
Admitted.

Theorem complement_is_function : forall (T : Type) (j k : cardinality)
        (proof: j + k = |T|), function (complement T j k proof).
    intros T.
    intros j k.
    intros proof.
    intros x y y'.
    intros proof_f_x_y proof f_x_y'.
    refine (subset_eq T k y y' _).
        intros a.
        destruct (complement T j k proof x y) as [proof_or proof_not_and].
        destruct (complement T j k proof x y') as [proof_or' proof_not_and'].
        unfold iff.
        refine (conj _ _).
            intros proof_in_y.
            pose (proof_not_in_x (proof_in_x : x a) := proof_not_and (conj proof_in
            case proof_or'.
                intros proof_in_x.
                case (proof_not_in_x proof_in_x).

                intros proof_in_y.
                exact proof_in_y.
```

```
            intros proof_in_y'.
            pose (proof_not_in_x (proof_in_x : x a) := proof_not_and' (conj proof_i
            case proof_or.
                intros proof_in_x.
                case (proof_not_in_x proof_in_x).

                intros_proof_in_y'.
                exact proof_in_y'.
Qed.
```

I'm excluding the proof of injectivity because it's the same proof with the roles of x and y switched.

The next problem came when trying to prove totality, aka that for any x, there exists a y such that x and y are related. The way I went about doing this, I needed to prove that the disjoint sum of two sets x and y such that y contained all elements of T not in x had cardinality equal to T, which required finding another bijection, this one between the disjoint sum $x + y$ and the set T. This definition is below:

```
Definition disjoint_bij (T : Type) (s1 : T -> Prop) (a : T) (b : sig s1 + sig ~s1)
match b with
| inl (exist a' _) => a = a'
| inr (exist a' _) => a = a'
```

Starting with the proof that this was a function, I needed to axiomatize equality for another class of objects, and all of that is below:

```
Lemma exist_eq (T : Type) (s1 : T -> Prop) :=
forall a : T, forall proof1 proof2 : s1 a, exist a proof1 = exist a proof2

Theorem disjoint_bij_is_function : forall (T : Type) (s1 : T -> Prop), function (di
intros T.
intros s1.
intros x y y'.
intros proof_f_x_y proof_f_x_y'.
case y.
intros ea.
destruct ea as [a proof_a_s1].
simpl in proof_f_x_y.
case y'.
intros ea'.
destruct ea' as [a' proof_a_s1'].
simpl in proof_f_x_y'.
rewrite (thm_eq_trans (thm_eq_sym proof_f_x_y') proof_f_x_y)).
exact (exist_eq T s1 a proof_a_s1 proof_a_s2).

intros ea'.
destruct ea' as [a' proof_not_a_s1'].
```

```
simpl in proof_f_x_y'.
rewrite (thm_eq_trans (thm_eq_sym proof_f_x_y') proof_f_x_y)).
case (proof_not_a_s1' proof_a_s1).

intros ea.
destruct ea as [a proof_not_a_s1].
simpl in proof_f_x_y.
case y'.
intros ea'.
destruct ea' as [a' proof_a_s1'].
simpl in proof_f_x_y'.
rewrite (thm_eq_trans (thm_eq_sym proof_f_x_y') proof_f_x_y)).
case (proof_not_a_s1 proof_a_s1').

intros ea'.
destruct ea' as [a' proof_not_a_s1'].
simpl in proof_f_x_y'.
rewrite (thm_eq_trans (thm_eq_sym proof_f_x_y') proof_f_x_y)).
exact (exist_eq T s1 a proof_not_a_s1 proof_not_a_s2).
Qed.
```

This relation is not symmetric, so proving injectivity was not the same as proving that the relation was a function; instead, thankfully, it was much easier:

```
Theorem disjoint_bij_is_injective : forall (T : Type) (s1 : T -> Prop), injective
intros T.
intros s1.
intros x x' y.
intros proof f_x_y proof_f_x_y'.
case y.
intros ea.
destruct ea as [a _].
simpl in proof_f_x_y.
simpl in proof_f_x_y'.
exact (thm_eq_trans (proof_f_x_y (thm_eq_sym proof_f_x_y'))).

intros ea.
destruct ea as [a _].
simpl in proof_f_x_y.
simpl in proof_f_x_y'.
exact (thm_eq_trans (proof_f_x_y (thm_eq_sym proof_f_x_y'))).
Qed.
```

Then came totality. This one sounds like it wouldn't even be too hard to prove in Coq: my relation was a function from T to $x + y$, where x was defined as a proposition on elements of T and an element of $x + y$ is either inl a where a satisfies x or inr a where a does not satisfy x. Therefore, for any $a \in T$, just prove existence by giving the result: if a satisfies x, then give inl a as an example; otherwise, give inr a as an example.

The problem here is that Coq is based on constructivist logic, meaning that it is not necessarily true that a satisfies x or a does not satisfy x, which makes it completely impossible to prove totality of my bijection. There should be a workaround like with axiomatizing equality, but I haven't found a suitable one yet.

So, in summary, trying to prove $\binom{n}{k} = \binom{n}{n-k}$ in Coq took way longer than I expected and there were unforeseen barriers, but in spite of that, I don't see anything that suggests that Coq is completely unworkable for my project. Considering that the ultimate goal is for Coq code to be generated, the length of these proofs isn't a huge deal, and I will find a workaround for that one issue. One thing I did consider was moving away from the exact approach that Haven took and instead working solely with sets of natural numbers; Haven did what he did partly to avoid requiring functions to be computable, but that shouldn't be relevant to my project. Regardless, I'm confident that I will be able to make Coq work for what I want it to do.

Finally, onto my year-end goal statements.

For my A goal: I will have been able to programmatically convert a combinatorial proof written in English into Statement, Quantity, MathObject, and Intent data structures representing the structure in the proof, find implications between Statements and represent those in an additional data structure; write in Coq a foundation of code that can be used to more easily prove combinatorial statements in Coq, likely heavily relying on Andrew Haven's work in his master's thesis in order to do so; and finally, be able to programmatically convert from the data structures I created based on the English proof to an output proof in Coq which could then be verified using the Coq compiler. I will also create an application with a UI that will allow a user to select a text file containing a combinatorial proof written in English, then allow them to select a location and name for an output file, then write the Coq proof to that output file or indicate that a Coq proof could not be generated, possibly because the input proof correct. I will also create a GitHub repo containing my code in Kotlin dealing with the conversion of the English proof to intermediate data structure to an output Coq proof, as well as foundational code in Coq, both that written by me and by Andrew Haven, and my journals as well as example proofs I used in my project. The GitHub repo will also include instructions for installing and running my program with the UI. I will also give a presentation at TJ Star clearly explaining my goal, work done and approaches used throughout my project, as well as my results and the extent of my success in achieving my goals of translating a combinatorial proof in English to Coq. I will finally produce a paper more specifically explaining my work at every step of my project, including my justification for moving away from Stanford CoreNLP and the strengths and weaknesses of my approach in translating proofs in English to Coq as well as the Coq language in expressing combinatorial proofs.

For my B goal: I will have been able to programmatically convert a combinatorial proof written in English into Statement, Quantity, MathObject, and Intent data structures representing the structure in the proof, find implications between Statements and represent those in an additional data structure; write in Coq a foundation of code that can be used to more easily prove combinatorial statements in Coq, likely heavily relying on Andrew Haven's work in his master's thesis in order to do so; and finally, be able to programmatically convert from the data structures I created based on the English proof to an output proof in Coq which could then be verified using the Coq compiler. I will also create a GitHub repo containing my code in Kotlin dealing with the conversion of the English proof to intermediate data structure to an output Coq proof, as well as foundational code in Coq, both that written by me and by Andrew Haven, and my journals as well as example proofs I used in my project. I will also give a presentation at TJ Star clearly explaining my goal, work done and approaches used throughout my project, as well as my results and the extent of my success in achieving my goals of translating a combinatorial proof in English to Coq. I will finally produce a paper more specifically explaining my work at every step of my project, includ-

ing my justification for moving away from Stanford CoreNLP and the strengths and weaknesses of my approach in translating proofs in English to Coq as well as the Coq language in expressing combinatorial proofs.

For my C goal: I will have attempted to write code that convert a combinatorial proof written in English into Statement, Quantity, MathObject, and Intent data structures representing the structure in the proof and find implications between Statements as well as represent those in an additional data structure. I will also have attempted to create an application with a UI that allows a user to make use of my attempted research. I will also create a GitHub repo that contains code that I wrote as part of my project. I will also give a presentation at TJ Star explain my goal, work done, and results, as well as a paper going into more specific detail on my attempted research.

These statements may have been a bit long. Additionally, B ended up essentially being the same as A but without the UI, which doesn't seem right but I'm not sure how else to change it. I tried to make a reasonable C; it ended up mostly being a more vague description of a goal. Hopefully they work well enough as starting points.