ITP(CGE112): Course Report

- Akshay Naik, Devendra Reddy

Course Activities

August - November 2018

We started by studying Intuitionistic Logic, and then we explored a bit of different systems (*F** and *Coq*), and settled with *Coq*, as we felt it had more learning resources (at that juncture). Next, we moved on to study the basics of the *Coq* system from the *Software Foundations: Logical Foundations* book. At this point, we started to implement the congruence closure algorithm in the *Coq* system. Congruence Closure formalisation has a similar high level structure as the regular expressions formalisation described in the *Software Foundations* book, although the former is much more intricate.

January - February 2019

We attended lectures of the 2-credit *Interactive Theorem Proving* course, and gave in submissions to both the *Coq* assignments that were part of this course, whilst continuing to work on the original course project.

Course Project: Verification of congruence closure.

Summary of attempts

We implemented the congruence closure algorithm for terms with functions of fixed(1) arity using a list of pairs(term*term) to store the representative for each term that occurs in the list of equalities and tried to formally verify it in *Coq*. However we failed to prove crucial properties about our representation that were necessary to prove the main soundness and completeness theorems. As an intermediate step to try and achieve the final goal, we tried restricting the theorem statements to only transitive closure by erasing the congruence rule in the inductive relation for `proof`.

Concurrently, we also did a rewrite in which we reworked the representation for transitive closure by using sets of elements as representatives. We hoped that using the *Coq* standard library's implementation of sets(using lists) would be more conducive to proving the required properties since many properties about set operations are available in the standard library. However, this approach too quickly grew in complexity and we ended up spending way more time than we had anticipated in proving useful invariants about the operations performed by our program. More than halfway into this endeavour it was difficult to see how one could demonstrate completeness of this procedure.

Following our presentation in July 2019, where we were advised to assume the representation and prove the main closure computing algorithm, we did a rewrite using dependent types to express the correctness properties of the transitive closure computing algorithm (`do_tc` in file `transitive_closure_submission.v`) and proceeded to **program** the same **using tactics**, thus resulting in a program that is **correct by construction**. To achieve this, we assumed useful properties about our representation by using(Admitted) an appropriately typed `merge` operation. We later extended the same approach to work for the congruence closure algorithm(`do_cc` in file `congruence_closure_rewrite_suffix.v`). As part of these efforts, we read relevant sections from *Adam Chlipala*'s book: *Certified Programming With Dependent Types* and the *Coq standard library*.

Brief explanation of proof strategy in final submission of Transitive Closure

Basic definitions

(Refer: `transitive_closure_submission.v`)

We will explain the overall proof strategy using transitive closure since it remains the same for congruence closure, but is concerned with slightly fewer details. It might be useful to interactively move through the corresponding source file whilst reading this description.

(For most of the definitions that occur in this report, in the corresponding source you will notice the lines following these definitions usually have various useful properties proved about them as `Lemma`.)

Each term is represented as a constructor taking a *nat*(We deal with function symbols later in *congruence closure*).

```
Inductive term : Set :=
  | var : nat -> term.
```

Over these terms we define an inductive `proof` relation that describes how a proof of equality between any two terms may be built. Note that `l` is the list of equalities provided to us, with respect to which we compute the closure under these rules(you will see this referred to as `eql` later in the source).

```
Inductive proof (l : list (term*term)) : term -> term -> Prop :=
  | proofAxm : forall s t, In (s, t) l -> proof l s t
  | proofRefl : forall t, proof l t t
  | proofSymm : forall s t, proof l s t -> proof l t s
  | proofTrans : forall s t u, proof l s t -> proof l t u -> proof l s u.
```

We achieve this by building a procedure: `do_tc` using tactics. One can think of the proof of `do_tc` as being a program which constructs a witness of the return type of `do_tc`. The type of this witness ensures that this procedure is **sound** and **complete**. Before we dive into the details of the *type* and *proof* of `do_tc`, we shall define a few auxiliary *predicates* and *methods*.

First, let's define the type for the map that we'll use to store the representative for each term in our list.

```
Definition mapRep (R:{T: Type | Decidable_Eq T }) := term -> (proj1_sig R).
```

We restrict the type of the representative `R` to be of a type having decidable equality. (After looking up the representatives of two terms, we need to be able to tell if they're the same or not.)

Then we go on to define an auxiliary predicate, **Well-Formed-Map(WFM)**. All the maps we deal with in `do_tc` will be **well-formed**. **WFM** is used ensure the soundness of the resulting map i.e. if two terms have the same representative, then we are assured to have a proof of their equality.

```
Definition WFM (eql: list (term*term)) (R:{T: Type | Decidable_Eq T}) :
  mapRep R -> Prop :=
    fun ufm =>
        (forall t1 t2, ufm t1 = ufm t2 -> proof eql t1 t2).
```

Next, we assume a proper representation by assuming the existence of a `merge` operation with the following type:

```
Definition merge R eql
(a b: term) (Hpf: proof eql a b)
  (ufm: { m: mapRep R | WFM eql R m }) :
      ({ m: mapRep R | WFM eql R m /\
        (forall x, (proj1_sig ufm) x = (proj1_sig ufm) a ->
                    m x = (proj1_sig ufm) b) /\
          (forall x, (proj1_sig ufm) x <> (proj1_sig ufm) a ->
                    (proj1_sig ufm) x = m x) } ).
Admitted.
```

Note that `ufm` is the input map(from terms to their representatives) which `merge` modifies.
To merge two terms `a` and `b`, `merge` also takes a proof of their equality, thus a merger of equivalence classes(via change of representatives) using a call to `merge` is assured to be justified. In the return type of `merge`, the first predicate in the conjunct, `WFM eql R m` assures that the resulting map is sound, and the second predicate ensures two things:

1. For all the terms `x` that had the same representative as `a` in the old map, will now have whatever representative `b` had in the old map. i.e. Equivalence class of `a` is merged with equivalence class of `b` by changing representatives.
   Simplified: `forall x, ufm x = ufm a → m x = ufm b`
2. For all the terms `x` that were **not** in the equivalence class of `a`, we leave their representatives untouched.
   Simplified: `forall x, ufm x ≠ ufm a → ufm x = m x`

While the above definition of merge is sufficient to prove transitive closure, it's not strong enough to actually implement merge itself. In the source, we provide a further strengthening of the type of merge(`line 276`) which ought to allow correct implementation.

Lastly, we have the following two auxiliary definitions to aid us in expressing the correctness of `do_tc`.

```
(* t occurs in eql *)
Definition Occurs (eql: list (term*term)) (t: term) :=
  exists x, In (t,x) eql \/ In (x,t) eql.
```

```
(* a is suffix of b *)
Definition suffix {A} (a b: list A) := exists c, b = c ++ a.
```

Finally, we're ready to state the soundness and completeness conditions for `do_tc`.
(decProc is a decision procedure for the representative type `R`, `eql` is the original list of equalities.)

```
Fixpoint do_tc {R} (decProc: DecEqT R) (eql: list (term*term))
  (l: {k: list (term*term) | suffix k eql}) :
    {m: mapRep R | WFM eql R m} ->
      {m: mapRep R | WFM eql R m /\
        forall a b, proof (proj1_sig l) a b -> m a = m b}.
```

Note that we use a slightly convoluted design for `do_tc` i.e. we keep around the original list of equalities `eql` and use its suffix `l` to induct over. This was deemed necessary to make it easier to prove soundness. Initially, `eql = l`, i.e. to compute the transitive closure over a list `eql` one would simply compute a lookup map(`final_rep_map`) as given below. The set of all terms having the same representative in this map is to be interpreted as being a single equivalence class.

```
final_rep_map = do_tc decProc eql eql init_map
(* Here, init_map would map each term in eql to a unique representative,
thus each term is initially in its own equivalence class. *)
(* Note that parameter R of do_tc is implicit. *)
```

In the return type of `do_tc`, `WFM eql R m` ensures that the resulting map `m` is **sound**. i.e. if two terms have the same representative then we have a proof of their equality. The second predicate ensures that for any two terms for which `proof l a b` holds, they will end up having the same representative. Since, initially `eql = l`, this ensures that `do_tc` is **complete**.

Proof strategy

The overall strategy is to induct over the suffix `l` and use the result of the recursive call to `do_tc` to build the appropriate witness by using merge. Devoid of all type information, this is basically what we're doing:

```
(* Pseudocode *)
do_tc eql ((a,b):l) ufm := merge R eql a b (proof eql a b) (do_tc eql l ufm)
```

The overall structure is best understood by interactively looking at all the intermediate assertions(and embedded comments) in the proof in the original source; especially `HMrecPf` on `line 237` and `Hfinal` on `line 255`.

Extending proof to Congruence Closure

(Refer: `congruence_closure_rewrite_suffix.v`)

The definitions and proof structure outlined above extends readily to a proof of congruence closure(with some augmentation of types to ensure correctness).

The major changes are mentioned below.
We add constructors for function symbols and the congruence proof rule.

```
Inductive term : Set :=
  | var : nat -> term
  | fn : nat -> term -> term.

Inductive proof (l : list (term*term)) : term -> term -> Prop :=
  | proofAxm : forall s t, In (s, t) l -> proof l s t
  | proofRefl : forall t, proof l t t
  | proofSymm : forall s t, proof l s t -> proof l t s
  | proofTrans : forall s t u, proof l s t -> proof l t u -> proof l s u
  | proofCong : forall (n : nat) s t, proof l s t -> proof l (fn n s) (fn n t).
```

Occurs is modified to allow for the existence of subterms.

```
Definition Occurs (eql: list (term*term)) (t: term) :=
  exists a b, In (a,b) eql /\ (Subterm t a \/ Subterm t b).
```

At face value, the correctness conditions for `do_cc` and `merge` remain the same. But they are slightly altered by the addition of one more condition to the *Well-Formed-Map* predicate as follows.

```
Definition WFM (eql: list (term*term)) (R:{T: Type | Decidable_Eq T}) :
  mapRep R -> Prop :=
    fun ufm =>
        (forall t1 t2, ufm t1 = ufm t2 -> proof eql t1 t2) /\
        forall x y n, ufm x = ufm y -> ufm (fn n x) = ufm (fn n y).
```

The change is justified as we now have to merge not only the equivalence classes of `a` and that of `b` but also that of their parents. i.e. if two arbitrary terms, `a` and `b` are(pair-wise) in the same class then so are `fn n a` and `fn n b` for all `n`.

The above changes allow us to use the same proof structure as in *transitive closure* and easily prove *congruence closure*.

Conclusion

We present a *formally verified, correct by construction* implementation(modulo the representation) of the *congruence closure* algorithm in *Coq*.

Thoughts, in hindsight.

Although *Coq* is an advanced variant of an interactive theorem prover, proving useful things(without pain) using its type system requires careful thought and clear abstractions.
Abstractions are more readily usable when algorithms are designed as compositions of well-typed functions. This allows one to reason locally and assret pre-conditions as types of the consisting functions.
On the whole, although this course project seemed to take up a lot of our time and at times it seemed we wouldn't be able to proceed without hands-on help from an an expert; it was a good learning experience which we hope will be useful in the future.