This time I do not have too much questions; instead, I think it's a good time to report something on my proof. I have started my proofs and finished the axomization and some lemmas. The source code can be found here[4][1].

Topics are two this time:

1. Some technical decisions I made when setting up proofs;
2. The main part would be just a showoff of what I did on proof engine engineering, by exploiting hint bases, auto tactic family, and other Coq features that have not been used in previous proofs that significantly reduce the burden.

1 Technical Decisions

1. I have strongly exploited unicode representation.
2. One problem I had previous was how to axiomize object only, in such a way that I will be very easy to use, which requires the axioms strong; while it needs to be easy to prove, which requires the axioms weak.

   I somehow find a midpoint, which effectively asserts objects have map semantics. In order to achieve this, I implemented a dependent data type, which is somewhat a stronger representation of list containment than `List.In`. One example is `subtyp_fld` rule for `subtyp` rules[4, Definition.v, 421].

   So far it hasn't bit back, but I am not entirely certain about that, because there isn't a way to prove from `Prop` to `Type`, and if it happens, there is no way out other than reverting this piece. In that case, we can only say something about the head of the `list`, and prove a sequence of lemmas to show that the subtyping rules allow binding to be anywhere in the `list`.

   On contrary, the benefit is also appealing: we now have an axiom that allows a binding appear literally anywhere in a `list`, as long as an witness is provided. Therefore we at least save lots of effort by not having to prove all those equivalence lemmas now, and given such witness, we can then define a very precise replacement function, as shown in the same reference rule.
3. Instead of TLC, I am using Metalib for this proof now[3].[2] From the feeling I got, I wouldn't say too good about it; there are pros and cons regarding this library and TLC.

   One problem on TLC that I feel strongest, is its over-encapsulation, introduced by Coq's module system. The module is opaque and therefore so are the types from the module, even though we know it for sure it's `list` or other stuff, we cannot just use the standard gears from Coq to reason about it, which is no doubt a pain, and Metalib beats this part.

   However, Metalib is not quite close to what it sounds to be. I can summarize this library has following issues.

   (a) The worst problem I can notice, is the library has heavily polluted the default hint base `core`. The consequence is, for some type of lemmas that are not true, `auto` and `eauto` might take unbearable amount of time to come back. This means this library is not well engineered; on contrary, TLC does a great job on that.

   (b) Secondly, this library is not designed for automated proofs. Lots of tactics it defines requires one to mention a lemma's name or so. Therefore, I have to reinvent something even if they already have a tactic for it, in order to maximize the capability of automated Coq proofs.

   Though, there are pros as well,

   (a) The library defines open ended modules and module types which allows certain level of reuse; however, it's not coded in the best way. If I had enough time to re-engineer library of such kind, I would base off my development on type class[3], so that the library design is completed open ended and support any extensions. I also have some issue with their associative list portion, but it's something can be got over with.

   (b) The library does have good support on cofinite quantification, tactics on finite set, and associative list.

[1] make will only compile those files being depended on, so some files needs to run coqc directly; I haven't got chance to take a look at makefile

[2] Interestingly, it's going to be unpleasant if one tries to work with two libraries together; these libraries both define the same notation, and therefore Coq will halt immediately at that point.

[3] which we will discuss later and it's used in my proof as well.

2 Proof Engineering

Following I will describe what did I do with my proofs to make most of the proof bodies virtually nothing. Proof automation is considerably more important in languages like DOT with mutual recursive definitions, which means there will be lots of repetitive property and function declarations afterwards, and it's observed in DOT proof. My intention of conducting such proof engineering beforehand, is in the hope of significantly reducing my maintenance effort in the case of small twists to the languages.

The leading philosophy is to achieve big picture proof by maintaining automation, tightly following the idea from [1]. Really for majority of the proofs, details like `subst` or similar are not interesting, or even close to line noise; while if the system is not able to figure it out by itself, the chances are some interesting undecidability can be found at that spot. I think I will explain what I did in three aspects.

2.1 Type Classes

It turns out there are type classes in Coq[5][2]. The feeling I have, is type classes are going to replace the classical module system. Type class is very nice at least in one way: all similar functions can be overloaded now. The consequence is, there is only one `open_rec` now. Coq will figure out by itself what should be the implementation when it's called. Since it's just a wrapper call, it can be easily unfolded by `simpl` evaluation. It's the same case for substitution and free variables. If anything else is needed, the same thing applies.

Once it's we achieve this, the next thing can be done is canonical way to express lemmas. In my current proof, it's no longer a pain to express lemmas anymore: it's always stated inside of `Notation` and then the next thing is just to repeat the notation to state the lemmas[4].

Another use of type class in Coq, has something to do with a tactic called `typeclasses eauto`[2, p. 20.6.5]. A rough understanding is this tactic is responsible to solve type class constraints, meaning it must be at least powerful enough to solve first order existential constraints. I have not had chance to understand those tactics and see how we can use them into the proofs, but I am pretty sure it can be very helpful, as I've seen people was talking about it and saying how useful it is.

2.2 Hint Bases

Another attempt I tried was to manipulate hint bases. I do not only add constructors and lemmas into hint bases now, but also add logic into it by mainly adding `Hint Extern` commands. When it's done properly, we can augment the power of `auto` family by doing more complicated things than it's originally defined. Another benefit is also it provides an outstanding description of "what kind of oracle is needed to finish this proof".

2.3 Writing Tactics

I also tried to aggressively write more tactics to assist the theorem proving. I tried to write my tactics in either of following two ways in order to work with the philosophy of automation:

1. The tactic itself should not mention any names from context or lemmas. If the purpose of such tactic is clear, the tactic is responsible for searching the current context to find out where itself can be applied to, to move the proof forward.

2. If we have to mention some names, the tactic needs to have enough search power so that we just need to mention the names of the lemmas but no more.

In `LibUtils.v`, I wrote a number of tactics that do following things:

1. (`routine`) Global master tactic;

2. (`cofinite`) Cofinite set instantiation;

3. (`reassoc`) List appending re-association;

4. (`exexec`) recursive tactic application by unfolding conjunctions and implications.

And some tiny others. You might also see `boom` somewhere, it's my personal naming convention for local master tactic.

There are still many things can be done on the tactic's side. Effectively, all observed proof pattern can be converted into a tactic to address certain routine behavior of the system. In that sense, writing more compact tactics itself helps to express the big picture of the proofs.

[4]e.g. `https://github.com/HuStmpHrrr/dot-calculus/blob/alg/src/simple-proof/alg-proof/Substitution.v`

3 Some Quirk in Current Axiomization

I want to briefly mention something strange in current definition. Curiously, we need to have a special treatment for opening object types and object values (`open_blah_to_context` functions). This is due to the normal opening will not be able to open the 0 index which is pointing to the object itself, while when we push objects to the context, we will need to open those 0 indices.

So far I haven't seen something bad happen due to this, but I think it worth to mention it to you.

References

[1] *Certified Programming with Dependent Types*. Link.

[2] *Chapter 20 Type Classes*. Link.

[3] *metalib*. cf0e9f8ff0a1f918fd9829d9c49dcd0dd9142e67. Link.

[4] *proof source*. Link.

[5] Matthieu Sozeau and Nicolas Oury. "First-Class Type Classes". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-71067-7.