

Formalising the Soundness & Completeness of Propositional Logic

1 Basic Definitions

I started by defining an inductive datatype `Form` to represent the syntax of propositional formulas. The syntax is summarised by the following grammar, where the symbols used are defined in the code using the `notation` command. The datatype and notations are defined in the `src/formula.lean` file.

$$A, B ::= \{x\} \mid \perp \mid \sim A \mid A \wedge B \mid A \vee B$$

The variables as represented by x above are taken to be elements of some arbitrary type which has to be specified, and hence `Form` is parameterized by this type. For example, `A : Form vars` means A has the type of formulas with propositional variables `vars`. To avoid clutter, an ambient lean variable is used for the remaining definitions so there is always a `vars` present in scope when defining theorems. I also ambiently assume `vars` is denumerable in files where this is important, such as in the proof of the completeness theorem. I opted to represent only a minimal functionally complete set of connectives, so there is no implication.

I also defined another inductive type `Deriv` to represent Gentzen-style natural deduction proof trees/derivations, included in the `src/natural_deduction.lean`. The type is a prop parameterized by a context of type set (`Form vars`), i.e. a set of formulas, and by the goal of type `Form vars`, i.e. a single formula. The `notation` for `Deriv` is $\Gamma \Vdash A$. As the notation suggests, this is supposed to represent "A is derivable from the assumptions in Γ by natural deduction". We say that Γ is consistent when it cannot be used to derive \perp , i.e. $\Gamma \not\Vdash \perp$.

$$\begin{array}{c} \frac{}{\Gamma, A \vdash A} (\text{Ax}) \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} (\wedge I) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} (\wedge E_1) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} (\wedge E_2) \\ \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} (\vee I_1) \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} (\vee I_2) \quad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C}{\Gamma \vdash C} (\vee E) \\ \frac{\Gamma, A \vdash \perp}{\Gamma \vdash \neg A} (\neg I) \quad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \perp} (\neg E) \quad \frac{\Gamma \vdash \perp}{\Gamma \vdash A} (\perp E) \end{array}$$

Figure 1: The natural deduction rules represented by `Deriv`

2 Semantics

The semantics of the formulas is based on an evaluation function `eval` that evaluates formulas to their truth values, for which I used the built-in `bool` type. This function is basically a representation of the truth table semantics. For `eval`, a function from the propositional variables `vars`

to `bool` has to be provided, and we call this valuation function a *model* of some formula if the formula `evals` to `tt` under the given valuation. This is represented by the two notations $\llbracket A \rrbracket_v$ and $v \models A$, where A is a formula and v is a valuation.

The latter notation is overloaded to represent models of sets of formulas, i.e. $v \models \Gamma$, as well as to represent entailment. A set of formulas Γ entails a formula A iff every model of Γ is a model of A . This is written as $\Gamma \models A$.

A set of formulas is satisfiable when it has a model. We can show that this is equivalent to saying that the set of formulas do not entail \perp . The proof follows quite easily by the definitions of entailment and the evaluation function, which formally is done by feeding the definitions to the `simp` tactic.

3 Soundness Theorem

The soundness theorem informally states that if a formula can be derived, then it is semantically true, with the contextual assumptions, it states that if $\Gamma \Vdash A$, then $\Gamma \models A$. The proof follows by straightforward induction on the structure of the derivation $\Gamma \Vdash A$ which Lean also handles quite well using the `induction` tactic. In each case, the proof essentially mirrors the derivation rule. For example, in the proof for the $(\perp E)$ case, the key tactic is `exfalso` which replaces the goal to show that A is entailed with the goal `false`. This is exactly what is expressed by the derivation rule (in reverse).

My takeaways from working on this proof (and the satisfiability equivalence of the previous section) is the use of `simp` to unfold the definitions such as of entailment and the evaluation function, and to simplify it to something workable at the same time. Additionally, I learned about the `case` tactic which is nice for structuring the proof to look like a proper inductive proof, as well as to rename the newly introduced variables such as the inductive hypotheses.

I also proved an equivalent form of the soundness theorem that says if Γ is satisfiable then it is consistent. It follows from a straightforward proof by contradiction.

4 Completeness Theorem

The completeness theorem states the inverse of the soundness theorem: if a formula is semantically true then it can be derived. However, for completeness, we actually prove its equivalent form that states if Γ is consistent then it is satisfiable, which in the code is represented by `completeness'`. The proof is based on the one found in the [Sets, Logic & Computation](#) textbook.

Informally, the proof attempts to construct a model of Γ using the set itself. This is done by completely extending Γ such that for every formula A , either A or $\sim A$ is in its completion Γ^* . However, it is important that the completion preserves consistency (otherwise, the task is trivial, just take the set of all formulas).

From the completion, we generate a valuation v^* that returns `tt` if the variable is true in Γ^* , and

ff otherwise. It is only when Γ^* is consistent that we can show v^* to be a model of any formula A iff A can be found in Γ^* in the first place. This final result means that v^* is a model of Γ iff Γ is a subset of Γ^* . But it is easy to see that the latter is true, since Γ^* is an extension of Γ , and so is a superset of Γ . Hence Γ is satisfiable by the model v^* .

The second half of the proof about constructing the model is fairly routine when formalised, although it did force me to think about parts of the proof where I had to use Lean's classical reasoning in order to assume the decidability of certain propositions, such as membership in Γ and its consistency. In the lemmas, this is done by adding decidable typeclass assumption at the start of the proofs/definitions. When I finally prove the completeness theorem, I learned to use the classical tactic which allowed the required decidability-related typeclasses of the lemmas being used to be inferred.

The first and more interesting half, known as Lindenbaum's lemma, is the construction of the completion, and the proof that consistency is indeed preserved. In the informal proof, a sequence of sets Γ_n is enumerated which extend Γ one formula at a time.

$$\begin{aligned}\Gamma_0 &= \Gamma \\ \Gamma_{n+1} &= \begin{cases} \Gamma_n \cup \{A_n\} & \Gamma_n \cup \{A_n\} \text{ is consistent} \\ \Gamma_n \cup \{\sim A_n\} & \text{otherwise} \end{cases}\end{aligned}$$

Here, an enumeration of formulas A_n is simply assumed to exist. For the formal proof I had to first assume that `vars` is denumerable and construct a bijection between formulas and the natural numbers, and also show that it is indeed a bijection. The constructions are conceptually simple but annoying to explicitly construct in Lean. For example, the function `Form.enc` that encodes formulas to natural numbers can be defined simply by structural recursion on formulas, but the corresponding inverse decoding `Form.dec` required me to learn about how to use strong/well-founded recursion on the natural numbers in Lean. This required me to have proofs that the numbers I'm passing into a recursive call is indeed less than the original input number. I also learned how to define a `match` expression on a natural number when it has been proven to be less than a particular value (i.e. to match on 0, 1, 2, 3 instead of the generic 0, $k + 1$ matching).

I struggled quite a bit with the proof of bijection because it required reasoning about the definition of `Form.dec` which as explained above is a `match` expression. In the Lean tactic state's goal, unfolding the definition leads to a massive desugared expression using the generated `dec._match_1` function which I didn't quite understand. `simp` didn't immediately work and I wasn't sure what to do. I discovered that `simp` can unfold `dec._match_1` only when its last argument, the value to be matched on, fit one of the patterns, which in my case is one of 0, 1, 2, 3. Hence, I realised that the general strategy to simplifying these horrifyingly large expressions was to first simplify its last argument.

While finding out about this strategy, I also learned about the `conv` tactic which allowed me to target specific parts of the goal, that is the last argument. In one case this helped avoid an error where the LHS of my `rw` lemma appeared in other parts of the goal, and using a general rewrite on all the instances caused a "Motive not type correct" error. Using `conv`, I `rw` only the `match` expression's last argument, which avoided the error.

A lot of the parts in proving the denumerability of Form vars felt repetitive and could probably be automated. There's probably opportunity here to implement a tactic that derives a denumerability instance for inductive data types.

With A_n defined, Γ_n can now also be defined. In the code, this is defined as the `lindenbaum_fn` function. We can now proceed to the construction of the completion Γ^* (`CΓ` in the code), which is simply the union of all the Γ_n . It is easy to show that Γ^* is indeed complete since for any formula A , we know it appears in the enumeration at some index k , in which case it (or its negation) must be included in Γ_{k+1} by definition.

Γ^* 's consistency is harder to prove. To begin with, we prove the lemma that each of the Γ_n are consistent by induction on n . This is expressed by the `lindenbaum_fn_consistent` in the code. While writing this proof I learned about the set tactic to assign shorter names to long terms in the goal. We then prove that the Γ_n form a chain, where if $i \leq n$ then $\Gamma_i \subseteq \Gamma_n$. Again this is proven by induction.

The second lemma allows us to show that all finite subsets of Γ^* are a subset of Γ_i for some i . Informally, this is easy to see as each formula in the finite subset is an element of some Γ_i . Since there's a finite amount of those, there are only a finite amount of Γ_i s, so by the second lemma the Γ_i with the highest index i must contain all the other Γ_i s, and hence the finite subset. The formal proof is more convoluted because we need to split into the case where the finite subset Δ is nonempty or empty. Only in the former case can the maximum be always well-defined. In the latter case, Δ is the empty set so it is trivially a subset of say Γ_0 .

Once we prove this, we can show that any finite subset of Γ^* is consistent since it is a subset of some Γ_i which by the first lemma is also consistent. From here, we have to prove and apply the syntactic compactness theorem, which states that if every finite subset is consistent then the whole set is consistent. Its contrapositive states that if $\Gamma \Vdash A$ then there is a finite subset Δ of Γ such that $\Delta \Vdash A$ also. This is quite easy to prove informally: a derivation is a finite structure, so it can only contain finitely many uses of the assumptions in Γ anyway. However to formally show this, I had to define an explicit construction of Δ which contains only the (undischarged) assumptions used in the derivation by induction. While this proof was not too difficult, it's yet another example of how a one-line, simple and intuitive informal proof can become a much more complicated formal proof. Because the theorem is a purely syntactic one, I placed it in `src/natural_deduction.lean` as `deriv_compactness` for the contrapositive and `deriv_compactness'` for the main result we need.

Having proven the alternate form of the completeness theorem, we can use it to prove the main form, which is the completeness theorem in the code.

Finally, I put everything together in `src/main.lean`, defining theorems that allow me to rewrite \Vdash to \vdash and `consistent` to `satisfiable`. We can use this to show the semantic version of the compactness theorem as a corollary by rewriting the \Vdash in the syntactic compactness theorem, as also demonstrated in this file.