Institut Polytechnique de Paris
Master 2 - Master Parisien de Recherche en Informatique

November 20, 2023

2.7.2 - Proof assistants

Natural Deduction, Heyting Semantics, and Proof by Reflection

TABET GONZALEZ Salwa

Previous experience with proof assistants

Even if I had taken the course in 2022/2023, I didn't learn much because of repeated absences and medical issues. For what it's worth, I obtained the grade of 14.45 on the past project, but I didn't get to do the section on reflection and defining our own tactic to prove tautologies. However, I obtained the grade of 5 on the final examination. Prior to that, I had never studied Coq: my masters 1 only had one course on logic where we occasionally used Agda, and my bachelor was more AI-oriented.

1 Natural Deduction

1.1 Intuitionistic

1.1.a nd

I defined the predicate `nd` as an inductive predicate, as required. At first I made a mistake on the definition of the assumption, having stated `s::A` as a premise, instead of `In s A`.

1.1.b Proofs of statements

Here I decided to construct proof terms instead of doing it in proof mode to apply what we had learned in lesson 4.

1.1.c Weakening

The proof of this was straightforward.

1.1.d Ground

At first, I had done a fixpoint construction for the `ground` predicate. However, it was a little difficult to unfold when working on ground decidability, so I just changed it into an inductive predicate with two constructors: one for $\bot$ and one for implication.

Then, I made a fixpoint for the function `ground_truth` which directly evaluates a ground formula to a boolean value. Then, I showed that a proof that a formula evaluates to `true` gives an intuitionistic deduction of this formula, and this was enough to deduce decidability of ground formulas from decidability of booleans.

1.2 Classical

1.2.a ndc

Again, I defined the predicate `nd` as an inductive predicate, as required.

1.2.b Proof of statement

I decided again to make a proof term to apply what we learned in lesson 4.

1.2.c Weakening

This proof was a straightforward induction on a predicate of type `nd A s`.

1.2.d Implication

This proof was easy: I just had to think of it as an induction on the type of proof of the nd term (hence `induction I`).

1.2.e Friedman translation

I did as requested: a function replacing every occurrence of `bot` by `t` and every occurrence of `var x` by `(var x ∼> t) ∼> t`. When the project handout states the need for a function, I automatically define a fixpoint.

1.2.f DNE_Friedman

The first thing I thought of was to make an induction on the structure of `s`. For the first case, in order to understand what was going on, I first thought of it as a quadruple negation implying a double negation. So, I made a lemma that was called `neg_4_2` (now named `imp_4_2`) that proves this case. The rest followed seamlessly.

1.2.g Friedman

Having proved `DNE_Friedman` before, this lemma was easy to prove.

1.2.h Ground formulas

As there are only two constructors of the predicate `ground`, the induction on this predicate has only two cases. The first one is proven by using the answer to question 1.2.d. For the second one, I made a lemma named `trans_ground` that made the development easier.

In this lemma, the first case is proven by mere reflexivity, while the other handled several equalities that could be solved by the `congruence` tactic. I decided to change my proof, made with other tactics, in order to show what I learned in lesson 6.

1.2.i Consistency

Intuitionistic natural deduction is not consistent if and only if $[]|-\bot$. Classical natural deduction, on the other hand, is not consistent if and only if $[]|-c\bot$. Since $\bot$ is a ground formula, we have shown that these two notions are equivalent.

1.3 Consistency

1.3.a Soundness

To prove this, I needed a lemma stating that the relation `leq` is transitive. This proof was easy as it was a simple proof by cases. Then, I proved the main theorem by cases too: at one point I have 27 cases to tackle, so I searched for tacticals that could help me (I had done this before lesson 6) and found `solve`.

1.3.b Double negation elimination

Now that we have proved soundness, everything follows.

1.3.c Intuitionistic consistency

Again, seamless proof.

1.3.d Classical consistency

Having proved consistency of intuitionistic natural deduction and that both intuitionistic and classical natural deduction derive the same ground formulas, it is easy to prove this theorem.

2 Heyting Semantics

2.a Extension with conjunction, and weakening

Adding conjunction as a constructor of the inductive predicate `form` was straightforward. Having done the first exercise, adding the corresponding

introduction and elimination rules was also quite easy. Weakening with these added rules was just as easy.

2.b Heyting algebras

Remembering what we had done in lesson 5, I just built a record with all the rules present in the handout inside.

2.c Evaluation

I figured the construction `meet` was equivalent to the conjunction, and the `impl` construction equivalent to the natural deduction implication.

2.d Meet_list

It wasn't specified in the handout if we had to evaluate the elements of the list, but since there is a valuation function passed as an argument I figured we had to. So, I defined a fixpoint.

When the list is empty, I first defined the return H as $\bot$. But while talking to a colleague, I understood that it had to be a neutral element for conjunction, so it had to be $\top$.

2.e Soundness

Again an induction on the rule applied to prove `s`. The proof in itself is quite long, and I tried finding ways of defining my own tactics to shrink it, to no avail.

2.f Revert

This revert function works just like the `revert` tactic: that is what helped me define it. Then, the correctness lemma is proved by induction on the structure of list `A`: this way I can work with the definition of the fixpoint `revert`.

2.g Formulas' Heyting algebra

First, I defined the relation `leq` as the implication, `bot` as $\bot$, `impl` as our implication, and `meet` as our conjunction. Then, I built a proof for every rule and finally constructed an instance of type `HeytingAlgebra` containing everything.

2.h Evaluation in this Heyting algebra

This proof was very straighforward and ressembles those made in class for lesson 2.

2.i Completeness and conclusion

Once we have shown that evaluation in the Heyting algebra of formulas is identity, it was straightforward to prove completeness, using this particular Heyting formula as a witness.

2 bis Heyting Semantics extended with disjunction and quantifiers

2.a Extension with disjunction and quantifiers, and weakening

Disjunction Adding the disjunction constructor to the type of formulas was straightforward, as well as adding the introduction and elimination rules corresponding to this constructor, which are the same as in classical logic :

```
| Idisjr A s t : nd A t -> nd A (disj s t)
| Idisjl A s t : nd A s -> nd A (disj s t)
| Edisj A s t u : nd A (disj s t)
                  -> nd A (imp s u)
                  -> nd A (imp t u)
                  -> nd A u
```

Bound and free variables In order to add constructors for the universal and existential quantifiers, I had to discern free variables and bound variables. The `var` constructor was replaced by a new constructor `free_var`, which behaves the same as `var`, and I added a new constructor `bound_var` for bound variables. With this paradigm, every occurence of a bound variable is represented by its *De Bruijn index*, which is the number of quantifiers on the path from said occurence to the quantifier which binds it in the formula tree. The new `form` type is as follows:

```
Inductive form : Type :=
| free_var (x : nat)
| bound_var (x : nat)
```

```
| bot
| imp (s t : form)
| conj (s t : form)
| disj (s t : form)
| for_all (s : form)
| exist (s : form).
```

Now, adding the introduction and elimination rules for these quantifiers requires the addition of :

— A new function `unbind : form → nat → form` which replaces all bound variables with De Bruijn index 0 with a given free variable.

— A new inductive predicate `not_free_in : nat → form → Prop` which guarantees that a given free variable does not appear in a given formula.

— A new predicate `not_free_in_ctx : nat → list form → Prop` which iterates `not_free_in` on a list of formulas.

For instance, the following deduction rule

$$\frac{A \vdash s[x]}{A \vdash \forall t, s[t]} \forall\text{-intro}$$

(which is conditioned by the fact that x does not appear as a free variable in A or s), becomes

```
| Ifor_all A s x :  not_free_in_ctx x A ->
                    not_free_in x s ->
                    nd A (unbind s x) ->
                    nd A (for_all s)
```

Weakening While extending the proof of `Weak` to this new logic, the constructor `disj` poses no issue. However, the quantifiers present a major obstruction. Notably, when performing induction on a term of the form `for_all s` (introduction rule `Ifor_all` above), the variable `x` could possibly appear in the larger context, making the obvious proof impossible. To solve this issue, I defined a function `exchange_vars` which takes two variables and swaps their occurences in a formula. After proving that exchanging variables in all nodes of a proof tree preserves its validity, I was able to perform induction in the `Ifor_all` case by exchanging the variable `x` with a new variable which does not appear in the larger context. I also had to use a similar solution for the elimination of the existential quantifier.

2.b Heyting algebras

In order to evaluate formulas in a Heyting algebra, this structure has to be expanded to be able to take the infimum of any definable subset of `H` (represented as predicates `H -> Prop`).

```
Record HeytingAlgebra : Type :=
{
  H : Type;
  leq : H -> H -> Prop;
  leq_refl s : leq s s;
  leq_trans s t u: (leq s t -> leq t u -> leq s u);
  bottom : H;
  inf : (H -> Prop) -> H;
  impl : H -> H -> H;
  leq_bottom u : leq bottom u;
  leq_inf (P : H -> Prop) u : (forall (s:H), P s -> leq u s) <-> leq u (inf P);
  leq_impl s t u : (leq (inf (finite_set [s;t])) u) <-> leq s (impl t u)
}.
```

Note that the `meet` operation does not need to be included in the definition, since it can be retrieved by taking the infimum of a set with two elements. Similarly, I defined a function for taking the supremum of a set, only using the infimum function.

I then proved all the basic properties of these functions, which correspond to the deduction rules in classical logic :

```
Lemma leq_meet {HA} (s t u : H HA) :
  (leq u s /\ leq u t) <-> leq u (meet s t).

Lemma leq_sup {HA} (P : H HA -> Prop) u :
  (forall (s : H HA), P s -> leq s u) <-> leq (sup P) u.

Lemma leq_join {HA} (s t u : H HA) :
  ((leq s u) /\ (leq t u)) <-> leq (join s t) u.

Lemma leq_impl_join {HA} (s t u : H HA) :
  leq (join s t) (impl (meet (impl s u) (impl t u)) u).
```

2.c Evaluation

Evaluating formulas by a direct induction was impossible, since I did not have a value to assign to bound variables. To circumvent this issue, I defined an inductive predicate `is_valid depth f` which, given a natural integer `depth`, asserts that every bound variable in `f` would be linked to a quantifier by adding `depth` quantifiers in front of `f`.

Now, after implementing the type of length-indexed lists, I was able to define a function `eval_rec`, which given a proof of `is_valid depth f` and a length-indexed list of length `depth` corresponding to the values to assign to bound variable, would evaluate a formula in a Heyting algebra.

2.d Meet_valid_ctx

Defining this function was straightforward once the previous function was done. The process was a same as for the function `Meet_list` in the first part of the exercise.

2.e Soundness

I encountered three major challenges while trying to prove soundness of Heiting semantics :

— The first challenge was that some proof nodes would involve formulas which do not appear in the context or the conclusion (for instance in the case of the elimination of implication). This was an issue since I would not have a proof of validity for these formulas. I circumvented this by taking a higher depth at which all formulas involved would be valid, evaluating the formulas at some extended length-indexed lists, and finally proving that the evaluation only depends on the first elements of the list.

— Secondly, for the proof nodes involving infimums and supremums, I was only able to prove functional equality, instead of structural equality of the predicates involved, which meant I could not rewrite some equalities. To solve this, I had to define the equivalence relation given by $u \sim v \Leftrightarrow u \leqslant v \wedge v \leqslant u$ in a Heyting algebra, and work with this relation instead of equality.

— Lastly, for instance in the case of introduction of the universal quantifier, I had to prove a correspondence between evaluation of a formula by

replacing a bound variable with a free variable, and evaluation of the initial formula.

2.f Conclusion

After proving soundness of Heyting semantics with general formulas, I specified my evaluation function to take only formulas which are valid, i.e. in which every bound variable is actually linked to a quantifier. I defined a simpler evaluation function for these formulas and proved soundness in this case.

```
Record valid_form : Type :=
{
  f : form;
  proof : is_valid 0 f
}.

Definition eval {HA} (val : nat -> H HA) (s : valid_form) : H HA :=
  eval_rec val (proof s) (Vnil (H HA)).
```

3 Reflection

3.a Disjunction

As done in the bonus exercise, I extended the `form` inductive predicate with a constructor `disj`, which takes as arguments two formulas.

3.b Transformation

I defined a fixpoint for this question. Then, to prove that evaluating a formula f into booleans gives the same result as evaluating its transformation into $\mathbb{Z}$, I had to define a fixpoint that evaluates the formula into booleans. I named it `evaluation`, as opposed to `reflection`. Finally, I could prove the theorem by mapping the results of the evaluation into $\mathbb{Z}$, and giving a valuation function to the reflection using 1 and 0.

3.c Proving boolean tautologies

This question was difficult. First, I decided to create a new reification tactic, called `reify2`. This tactic reifies the first formula, adding its variables to a list, and then reifies the second formula using the variables introduced

before. This ensures that each variable has the same index in both formulas: I identified this problem while trying to solve `andb a b = andb b a`.

I couldn't find a way to reuse the `reflection` function in the actual automatic proving tactic. Instead, I "redefined" reflection with lemmas, and used the `rewrite` tactic to unfold the formula obtained after reification.

In the rewriting tactic `rewrite_bool`, there is a `try` tactical before each `rewrite` tactic on a specific formula structure. I did this because the `rewrite` tactic rewrites all occurrences in the goal, but I only needed one specific occurrence. Instead of fighting against the usual behaviour, I just allowed the tactic to fail. I identified this problem while trying to solve `(andb b (andb a b)) (andb a b) = andb b a`.

3.d Completeness

This process cannot solve classical logic boolean equalities, such as `andb a (neg a) = false`, or the excluded middle `orb a (neg a) = true`. This is due to the unfolding of `bool_to_Z` into an equation that can only be solved by a case analysis on the value of `a`.