Type Theory and Coq 2019-2020

24-06-2020

Propositional logic:

1. Give a proof in first order intuitionistic propositional logic of the formula:

   $$(\neg a \vee b) \to (a \to b)$$

   (The opposite direction of the implication is also provable, but only classically. Proving that is not part of this exercise.)

$$\dfrac{\dfrac{[\neg a^z] \quad [a^y]}{\dfrac{\bot}{b}\, E\bot}\, E\to \quad \dfrac{[b^w]}{b \to b}\, I[w]\to}{\dfrac{[\neg a \vee b^x] \quad \dfrac{b}{\neg a \to b}\, I[z]\to}{\dfrac{b}{a \to b}\, I[y]\to}\, E\vee}{(\neg a \vee b) \to (a \to b)}\, I[x]\to$$

2. Give both the natural deduction proof that corresponds to the proof term

   $$(\lambda x : a \to a.\, x)(\lambda x : a.\, x)$$

   as well as the normal form of that proof.

$$\dfrac{\dfrac{[a \to a^x]}{(a \to a) \to a \to a}\, I[x]\to \quad \dfrac{[a^x]}{a \to a}\, I[x]\to}{a \to a}\, E\to \qquad \dfrac{[a^x]}{a \to a}\, I[x]\to$$

Simple type theory:

3. Give the most general type in simple type theory of the term:

   $$\lambda xyz.\, x(yz)zy$$

   You do not need to explain why this is the most general type.

$$\begin{array}{c}\lambda x : a \to b \to (b \to a) \to c.\, \lambda y : b \to a.\, \lambda z : b.\, x(yz)zy \\ : \\ (a \to b \to (b \to a) \to c) \to (b \to a) \to b \to c\end{array}$$

4. Give a type derivation in simple type theory that shows that

$$\lambda x : a \to b.\, \lambda y : b \to c.\, \lambda z : a.\, y(xz)$$

is a well typed term.

$\Gamma := x : a \to b,\ y : b \to c,\ z : a$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash y : b \to c} \quad \dfrac{\dfrac{}{\Gamma \vdash x : a \to b} \quad \dfrac{}{\Gamma \vdash z : a}}{\Gamma \vdash xz : b}}{\Gamma \vdash y(xz) : c}}{x : a \to b,\ y : b \to c \ \vdash\ \lambda z : a.\, y(xz)\ :\ a \to c}}{x : a \to b \ \vdash\ \lambda y : b \to c.\, \lambda z : a.\, y(xz)\ :\ (b \to c) \to a \to c}}{\vdash\ \lambda x : a \to b.\, \lambda y : b \to c.\, \lambda z : a.\, y(xz)\ :\ (a \to b) \to (b \to c) \to a \to c}$$

Predicate logic and dependent types:

5. Give a proof in first order intuitionistic first order predicate logic of the formula:

$$\forall x.\, ((\forall y.\, P(y)) \to (\exists y.\, P(y)))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[\forall y.\, P(y)^H]}{P(x)}\, E\forall}{\exists y.\, P(y)}\, I\exists}{(\forall y.\, P(y)) \to (\exists y.\, P(y))}\, I[H]\to}{\forall x.\, ((\forall y.\, P(y)) \to (\exists y.\, P(y)))}\, I\forall$$

6. Give a full type derivation of the λP type judgment:

$$a : * \vdash a \to a \to * : \Box$$

For the rules of λP see page 10.

You may write recurring subderivations only once, and replace the other occurrences with dots.

$$\dfrac{\dfrac{\dfrac{}{\vdash * : \Box}}{a : * \vdash a : *} \quad \dfrac{\dfrac{\dfrac{}{\vdash * : \Box} \quad \dfrac{}{\vdash * : \Box}}{a : * \vdash * : \Box} \quad \dfrac{\dfrac{}{\vdash * : \Box}}{a : * \vdash a : *}}{a : *,\ y : a \vdash * : \Box} \quad \dfrac{}{\vdash * : \Box}}{a : * \vdash a \to * : \Box} \quad \dfrac{}{a : * \vdash a : *}}{a : *,\ x : a \vdash a \to * : \Box}$$

$$a : * \vdash a \to a \to * : \Box$$

Second order propositional logic and polymorphism:

7. Give a proof in intuitionistic second order propositional logic of the formula:

$$(\forall c.\,((\forall a.\,(A \to c)) \to c)) \to (\exists a.\,A)$$

(The opposite direction of the implication is also provable, but proving that is not part of this exercise.)

Hint: it might be useful to instantiate the universal quantifier with the statement you are trying to prove.

$$\dfrac{\dfrac{[\forall c.\,((\forall a.\,(A \to c)) \to c)^x]}{(\forall a.\,(A \to \exists a.\,A)) \to \exists a.\,A}\,E\forall \qquad \dfrac{\dfrac{\dfrac{[A^y]}{\exists a.\,A}\,I\exists}{A \to \exists a.\,A}\,I[y]\to}{\forall a.\,(A \to \exists a.\,A)}\,I\forall}{\dfrac{\exists a.\,A}{(\forall c.\,((\forall a.\,(A \to c)) \to c)) \to \exists a.\,A}\,I[x]\to}\,E\to$$

8. In $\lambda 2$ we can define:

$$A \times B := \Pi c : *.\,((A \to B \to c) \to c)$$

Now we want a projection function:

$$\pi_1 : A \times B \to A$$

Define this function as a lambda term.

$$\pi_1 := \lambda z : A \times B.\, zA\,(\lambda x : A.\,\lambda y : B.\,x)$$

Inductive types:

9. In Coq we can define dependently typed lists of Booleans in which the parameter of the type is the length of the list, with type:

```
boollist_dep
     : nat -> Set
```

We want to use the names `nil` and `cons` for the constructors.

Give the inductive definition of this type in Coq.

```
Inductive boollist_dep : nat -> Set :=
  | nil : boollist_dep 0
  | cons : forall n : nat,
      bool -> boollist_dep n -> boollist_dep (S n).
```

10. In Coq can define dependently typed lists Booleans:

```
Inductive boollist_dep : nat -> Set :=
  | nil : boollist_dep 0
  | cons : forall n : nat,
      bool -> boollist_dep n -> boollist_dep (S n).
```

Now give a recursive definition using `Fixpoint` and `match` of a function:

```
trues
     : forall n : nat, boollist_dep n
```

The value of `trues n` should be a list containing `n` copies the Boolean `true`.

```
Fixpoint trues (n : nat) {struct n} : boollist_dep n :=
  match n return boollist_dep n with
  | 0 => nil
  | S m => cons m true (trues m)
  end.
```

In the old Coq version in ProofWeb the annotation

```
return boollist_dep n
```

is required to make this work, but in a modern Coq it is not. This means that the solution without this annotation is also a correct answer to the exercise.

Induction principles:

11. Give the dependent induction principle for the unit type, as defined by:

```
Inductive unit : Set := tt : unit.
```

and explain the structure of this principle.

```
unit_ind
     : forall P : unit -> Prop, P tt -> forall u : unit, P u
```

(i) (ii) (iii)

The principle consists of three parts:

(i) For all predicates on the inductive type...

(ii) ...if that predicate is closed under each constructor then...

(iii) ...the predicate holds on the whole type.

12. With an appropriate definition of a predicate `even`, we get as the induction principle:

```
even_ind
     : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, even n -> P n -> P (S (S n))) ->
       forall n : nat, even n -> P n
```

Now we would like to show that one is not even, so we have the goal:

```
~(even (S 0))
```

When we use inversion to prove this, internally a proof term will be constructed that contains a subterm:

```
even_ind (fun n => n = S 0 -> False) H1 H2
```

Give the type of this term, as well as the types of `H1` and `H2`. (These correspond to subgoals that internally will be generated by the `inversion` tactic, and then automatically processed by the `discriminate` and `injection` tactics.)

this term
```
     : forall n : nat, even n -> n = S 0 -> False
```
```
H1   : 0 = S 0 -> False
```
```
H2   : forall n : nat,
       even n -> (n = S 0 -> False) -> S (S n) = S 0 -> False
```

Short explanation of why this is relevant for inversion: the formula proved by *this term* has the form

```
... even n -> n = S 0 -> False
```

but this is equivalent to

```
... n = S 0 -> even n -> False
```

so we can use it to prove that

```
       forall n : nat, n = S 0 -> even n -> False
```

But

```
forall n : nat, n = S O -> ...
```

amounts to saying that the property holds for `S O` (like Henri Ford's *a customer can have a car painted any color he wants as long as its black*, we have here that *this property holds for any number as long as it's one*). So this is how inversion works internally.

Correspondences:

13. Under the Curry-Howard correspondence, what corresponds with the logical operators conjunction and disjunction? You do not need to explain why this is a natural correspondence.

    The product type and the disjoint sum type.

14. Discuss the correspondences and differences between sorts $*$ and $\Box$ of the systems of the lambda cube (like λP and $\lambda 2$) on the one hand, and the Coq universes Prop, Set and Type on the other hand.

    The sort $*$ corresponds to Prop and Set, and the sort $\Box$ corresponds to Type.

    Differences are:

    - There is not a single Type but an infinite list of universes $\mathsf{Type}(i)$. For this reason the sort $\Box$ does not have a type, but the Type universes *do* each have a type, because $\mathsf{Type}(i)$ has type $\mathsf{Type}(i+1)$.
    - The Coq universes are in a subtype relation to each other:

      $$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \ldots$$

    - The sort $*$ is impredicative, but Set is predicative (as are the Type universes).

Metatheory:

15. Given that we know that $\lambda 2$ satisfies Strong Normalisation and Subject Reduction, how can that be used to show that $\lambda 2$ is consistent? A type theory containing $\lambda 2$ is consistent when the type

    $$\mathsf{new\_false} := \Pi a : *.\, a$$

    is not inhabited.

    You may use that new_false does not have a normal proof (this is proved using a theorem about the shape of normal forms).

    Suppose that new_false was inhabited, so we had M : new_false. Then by Strong Normalisation we could reduce M to a normal form N, with a reduction path $M \twoheadrightarrow_\beta N$. Now because of Subject Reduction we then also would get N : new_false. But that would give us a normal proof of new_false, and we were allowed to use the fact that such a proof does not exists. Therefore we get a contradiction.

    It follows that new_false is not inhabited, and therefore that $\lambda 2$ is consistent.

16. In the Strong Normalisation proof of $\lambda\!\rightarrow$ we defined an interpretation function, that maps simple types to sets of untyped lambda terms:

$$\begin{aligned}[\![a]\!] &:= \mathsf{SN} \\ [\![A \to B]\!] &:= \{M \mid \forall N \in [\![A]\!].\, MN \in [\![B]\!]\}\end{aligned}$$

In this, SN is the set of all strongly normalising lambda terms.

Now one of the lemmas in the proof consists of two properties, proved with simultaneous induction on the structure of the type. These properties are:

(a) $xN_1 \ldots N_k \in [\![A]\!]$ for all variables x, types A and $N_1, \ldots, N_k \in \mathsf{SN}$

(b) $[\![A]\!] \subseteq \mathsf{SN}$

The exercise: write out the function type case of the proof of the first property. (You do not need to prove the first property for the atomic type case, nor do you need to prove the second property in either case.)

We need to prove that

$$xN_1 \ldots N_k \in [\![A \to B]\!]$$

given the Induction Hypothesis that both properties hold for A and B. This amounts to showing that

$$xN_1 \ldots N_k \in \{M \mid \forall N \in [\![A]\!].\, MN \in [\![B]\!]\}$$

or in other words that

$$\forall N \in [\![A]\!].\, xN_1 \ldots N_k N \in [\![B]\!]$$

Now because of the property (b) in the Induction Hypothesis for A we know that if $N \in [\![A]\!]$ then $N \in \mathsf{SN}$. But then the property (a) in the Induction Hypothesis for B gives us that for such N

$$xN_1 \ldots N_k N \in [\![B]\!]$$

Which is exactly what we need to prove.

MetaCoq:

17. An abstract description of a basic form of reflection has the following structure. One defines a type T, together with two functions:

$$\begin{aligned} f &: T \to \mathsf{bool} \\ C &: T \to \mathsf{Prop}\end{aligned}$$

and then proves a lemma:

$$f\_\mathsf{correct} : \forall x : T.\, fx = \mathsf{true} \to Cx$$

That way one can prove a goal of type Ct with the proof term

$$f\_\mathsf{correct}\ t\ (\mathsf{eq\_refl}\ \mathsf{true}) : Ct$$

in which the conversion rule is used to convert from $\mathsf{true} = \mathsf{true}$ to $ft = \mathsf{true}$.

Now in the MetaCoq paper an implementation of a tautology checker is described that follows this structure. There are types form, seq and result, and there are constants:

$$\begin{aligned} \mathsf{Valid} &: \mathsf{result} \\ \mathsf{valid} &: \mathsf{seq} \to \mathsf{Prop} \\ \mathsf{tauto\_proc} &: \mathsf{nat} \to \mathsf{seq} \to \mathsf{result} \end{aligned}$$

The first argument of the last function is 'fuel'.

The exercise: explain how the types and functions of this tautology checker map to the abstract description, and give the statement of the counterpart of the f_correct lemma.

We have the following mapping:

$$\begin{aligned} T &\mapsto \mathsf{seq} \\ f &\mapsto \mathsf{tauto\_proc}\ n \\ \mathsf{bool} &\mapsto \mathsf{result} \\ \mathsf{true} &\mapsto \mathsf{Valid} \\ C &\mapsto \mathsf{valid} \end{aligned}$$

The counterpart of f_correct is:

$$\forall n : \mathsf{nat}.\ \forall x : \mathsf{seq}.\ \mathsf{tauto\_proc}\ n\ x = \mathsf{Valid} \to \mathsf{valid}\ x$$

Actually, a lemma like this occurs in the MetaCoq paper on page 35 under the name of tauto_sound:

```
tauto_sound n s : tauto_proc n s = Valid → valid s
```

18. Here are some lines from the MetaCoq formalisation:

```
Module NoPropLevel.
  Inductive t := lSet |  Level (_ : string) | Var (_ : nat).
End NoPropLevel.

Module UnivExpr.
  (* npe = no prop expression, +1 if true *)
  Inductive t := lProp | npe (e : NoPropLevel.t * bool).
End UnivExpr.

Module UnivExprSet := MSetList.MakeWithLeibniz UnivExpr.

Module Universe.
```

```
  Record t := { t_set : UnivExprSet.t ;
                t_ne  : UnivExprSet.is_empty t_set = false }.
End Universe.

Definition ident := string.

Inductive name : Set :=
| nAnon
| nNamed (_ : ident).

Inductive term : Type :=
| tRel (n : nat)
| tVar (id : ident) (* For free variables (e.g. in a goal) *)
| tEvar (ev : nat) (args : list term)
| tSort (s : Universe.t)
| tCast (t : term) (kind : cast_kind) (v : term)
| tProd (na : name) (ty : term) (body : term)
| tLambda (na : name) (ty : term) (body : term)
| tLetIn (na : name) (def : term) (def_ty : term) (body : term)
| tApp (f : term) (args : list term)
| tConst (c : kername) (u : Instance.t)
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ind_and_nbparams: inductive*nat) (type_info:term)
        (discr:term) (branches : list (nat * term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat).
```

The module `MSetList.MakeWithLeibniz` represents finite sets as lists, so `UnivExprSet.t` is just '`list UnivExpr.t`'. De Bruijn variables count from zero.

You may imagine that you are in the `string` scope, so you can type strings as '`"a"`'. A one element list is written as 'x`::nil`'. You can write an element of a record type as

'`{| t_set := ... ; t_ne := ... |}`'

An appropriate proof of the `t_ne` field in a `Universe.t` is '`eq_refl false`'.

The exercise: give a term in `term` that corresponds to the polymorphic identity:

$$\lambda a : *. \lambda x : a. x$$

and a term in `term` that corresponds to its type:

$$\Pi a : *. a \to a$$

Define s as:

```
tSort {| t_set :=
            (UnivExpr.npe (NoPropLevel.lSet,false))::nil;
          t_ne := eq_refl false
       |}
```

then the two required terms are:

```
tLambda (nNamed "a") s
  (tLambda (nNamed "x") (tRel 0) (tRel 0))
```

```
tProd (nNamed "a") s
  (tProd nAnon (tRel 0) (tRel 1))
```