Type Theory and Coq 2019-2020

24-06-2020

This exam consists of 9 out of 18 exercises (therefore, the numbering is not consecutive). Every exercise is worth 10 points, the first 10 points are free, and the final mark is the number of points divided by 10. The mark for the exam will be rounded to an integral mark afterwards. The oral part of this exam determines in which direction we will round.

To hand in your work, you have to submit photos or scans of your *handwritten* answers in the Brightspace assignment where you got this exam PDF. You can submit multiple times, but only the last submission will be looked at. Write your name and student number on your first answer sheet for reference.

Write proofs, terms and types according to the conventions of Femke's course notes. Obviously this exam is open book, and you can also use your computer if you like. However, communication among students is not allowed, see the notice at the end of the exam.

If a technical problem occurs during the exam, send mail to `freek@cs.ru.nl` and we will call you back as soon as possible, to see what we can do.

Good luck!

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.)

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.

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.

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.

Predicate logic and dependent types:

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

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

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 7.

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

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.)

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

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.

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.

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`.

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.

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.)

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.

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.

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).

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

(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.)

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.

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:

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

And finally: *By taking the exam, the student declares that no plagiarism is or will be committed. If the lecturer has the suspicion that fraud has been committed, the student will be contacted. If needed, the case will be redirected to the Examination Board.*

Typing rules of λP

axiom

$$\frac{}{\vdash * : \Box}$$

variable

$$\frac{\Gamma \vdash A : s}{\Gamma,\, x : A \vdash x : A}$$

weakening

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma,\, x : C \vdash A : B}$$

application

$$\frac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

abstraction

$$\frac{\Gamma,\, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B : s}{\Gamma \vdash \lambda x : A.\, M : \Pi x : A.\, B}$$

product

$$\frac{\Gamma \vdash A : * \qquad \Gamma,\, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.\, B : s}$$

conversion

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{when } B =_\beta B'$$