

# Type Theory and Coq 2018-2019

11-06-2019

Write your name and student number on each paper that you hand in. This test has 6 exercises, with 30 subexercises. Every subexercise is worth 3 points, the first 10 points are free, and the final mark is the number of points divided by 10. Write proofs, terms and types according to the conventions of Femke's course notes. Good luck!

1. This exercise is about *simple type theory* and *propositional logic*.

(a) Give a proof in minimal propositional logic of the formula:

$$(a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$$

(b) Give the proof term in (Church-style) simple type theory of the proof from the previous subexercise, which is a lambda term with type:

$$(a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$$

(c) Give a derivation of the typing judgement of the term from the previous subexercise.

(d) Give the three typing rules of simple type theory.

(e) Give the most general type of the lambda term:

$$\lambda xyz. x(yxz)$$

You do not need to show that this is the most general type, or how you obtained it, just giving the type is sufficient.

2. This exercise is about *dependent types* and *predicate logic*.

(a) Give a proof that contains a *detour* in minimal predicate logic of the formula:

$$\forall x. ((\forall y. p(y)) \rightarrow p(x))$$

(Note: if you do not know what a detour is, or you cannot find a proof with a detour, you can get partial points for a proof of this formula without a detour.)

(b) Give the proof term in  $\lambda P$  of the proof from the previous subexercise. Use the type  $D$  for the domain that is being quantified over.

- (c) Give the normal form of the term from the previous subexercise. Explain your answer.
- (d) Give the full  $\lambda P$  typing judgement (i.e., including the  $\lambda P$  context) of the term in normal form from the previous subexercise.
- (e) What is the formula in minimal predicate logic that has as proof term:

$$\lambda H_1 : (\Pi x:D. px \rightarrow qx). \lambda H_2 : (\Pi y:D. qy \rightarrow ry). \lambda z : D. \lambda H_3 : pz. H_2 z (H_1 z H_3)$$

3. This exercise is about *polymorphism* and *second order propositional logic*.

- (a) Give a proof in minimal second order propositional logic of the formula:

$$\forall a. (a \rightarrow a) \rightarrow a \rightarrow a$$

- (b) Give the proof term in  $\lambda 2$  for the proof from the previous subexercise.
  - (c) Give a *different* proof term for the formula from the previous subexercises, where ‘different’ means that the  $\beta$  normal forms are different.
  - (d) Give the lambda term for the *polymorphic composition operator*, which, apart from type arguments, takes two functions  $f$  and  $g$ , and returns the composition  $f \circ g$ .
  - (e) Does there exist a  $\lambda 2$  term  $M$  with type  $A$ , such that  $MA$  is a well typed  $\lambda 2$  term too? If so, give an example. If not, explain why.
4. This exercise is about the typing rules of *pure type systems* and the *lambda cube*.

For the typing rules of the lambda cube, see page 7 of this exam.

- (a) Give a derivation in  $\lambda \rightarrow$  of the judgement:

$$a : * \vdash (\lambda x : a. x) : (a \rightarrow a)$$

- (b) Conjunction can be impredicatively defined as:

$$\lambda a : *. \lambda b : *. \Pi c : *. ((a \rightarrow b \rightarrow c) \rightarrow c)$$

List the systems of the lambda cube in which this term is typable. Explain your answer.

- (c) The systems of the lambda cube all satisfy the property of *subject reduction*. State what this means.

- (d) The systems of the lambda cube all satisfy the property of *strong normalization*. State what this means.

5. This exercise is about *inductive types* and *recursive functions*.

- (a) We want a datatype for binary trees with no further data at either nodes or leaves. Define an inductive type `tree` of type `Set` using Coq syntax for this datatype. An example of an element of this type might be:

```
Node (Node Leaf (Node Leaf Leaf)) Leaf
```

- (b) Give the type of the recursion principle `tree_rec` for the inductive type from the previous subexercise.
- (c) Define a function `count_leaves` that counts the number of leaves of a tree using `Fixpoint` and `match`. The count for the example tree should be four, as there are four `Leaf`s. Remember that the Coq type for natural numbers is called `nat`, and the function for addition on natural numbers is called `plus`.
- (d) Define the same function using `tree_rec`.
- (e) Define an inductive predicate

```
mirrors : tree → tree → Prop
```

that states that the first argument is the left-to-right mirror of the second argument. Note that this is not a function that mirrors its input, but a *relation* between two trees.

For example the following type should be inhabited:

```
mirrors (Node (Node Leaf (Node Leaf Leaf)) Leaf)
        (Node Leaf (Node (Node Leaf Leaf) Leaf))
```

- (f) Give a definition of Leibniz equality as an inductive predicate. (There are two different versions for this, depending on whether one of the arguments of the equality is a parameter or not. Both versions are a proper answer to this exercise.)

6. This exercise is about *guarded type theory*.

- (a) Consider the following Coq definitions:

```

CoInductive stream : Set :=
| cons : nat -> stream -> stream.

Definition f (x : nat) (s : stream) : stream :=
  cons x s.

CoFixpoint fold_f (s : stream) : stream :=
  match s with
  | cons x s' => f x (fold_f s')
  end.

```

Coq will accept this without complaint, as `f` is just another name of `cons`, and therefore `fold_f` just returns its input stream in a complicated way.

However, the following definition, in which now `f` is an arbitrary function, is not accepted by Coq:

```

CoFixpoint fold (f : nat -> stream -> stream)
  (s : stream) : stream :=
  match s with
  | cons x t => f x (fold f t)
  end.

```

Explain the problem with this definition.

- (b) Haskell (in which streams generally are called ‘lazy lists’) *does* allow the counterpart of the definition of `fold` from the previous exercise:

```

fold :: (Int -> [Int] -> [Int]) -> [Int] -> [Int]
fold f (x:t) = f x (fold f t)

```

Explain in what way Haskell differs from Coq, so that Haskell does *not* have a problem with this function.

We will now look at a guarded type theory (in Curry-style). The syntax of the types and terms and ‘clock contexts’ of this theory is:

$$\begin{aligned}
A &::= a \mid A \rightarrow A \mid \mathbb{1} \mid A + A \mid A \times A \mid \mu a.A \mid \triangleright A \mid \square A \mid \uparrow A \\
M &::= x \mid \lambda x.M \mid MM \mid \\
&\quad \star \mid \text{inl } M \mid \text{inr } M \mid \text{case } M \text{ of } x.M; x.M \mid (M, M) \mid \text{fst } A \mid \text{snd } A \\
&\quad \text{construct}_{\mu a.A} M \mid \text{primrec}_{\mu a.A} M \mid \text{fix } M \mid \\
&\quad \text{next } M \mid M \otimes M \mid \text{box } M \mid \text{unbox } M \mid \text{force } M \mid \text{up } M \mid \text{down } M \\
\Delta &::= \emptyset \mid \kappa
\end{aligned}$$

In this  $a$  and  $x$  are respectively type and term variables. We write **construct** instead of the more customary **cons**, because we use the latter already for the stream constructor.

This theory has many rules, of which the rules that are relevant for this exam are, for the types:

$$\frac{\Gamma \vdash_{\kappa} A \text{ type}}{\Gamma \vdash_{\kappa} \triangleright A \text{ type}} \quad \frac{\uparrow\Gamma \vdash_{\kappa} A \text{ type}}{\Gamma \vdash_{\emptyset} \square A \text{ type}} \quad \frac{\Gamma \vdash_{\emptyset} A \text{ type}}{\uparrow\Gamma \vdash_{\kappa} \uparrow A \text{ type}}$$

And for the terms:

$$\frac{\uparrow\Gamma \vdash_{\kappa} M : A}{\Gamma \vdash_{\emptyset} \text{box } M : \square A} \quad \frac{\Gamma \vdash_{\emptyset} M : \square A}{\uparrow\Gamma \vdash_{\kappa} \text{unbox } M : A}$$

$$\frac{\Gamma \vdash_{\kappa} M : A}{\Gamma \vdash_{\kappa} \text{next } M : \triangleright A} \quad \frac{\Gamma \vdash_{\emptyset} M : \square \triangleright A}{\Gamma \vdash_{\emptyset} \text{force } M : \square A}$$

$$\frac{\Gamma \vdash_{\emptyset} M : A}{\uparrow\Gamma \vdash_{\kappa} \text{up } M : \uparrow A} \quad \frac{\uparrow\Gamma \vdash_{\kappa} M : \uparrow A}{\Gamma \vdash_{\emptyset} \text{down } M : A}$$

In these rules, for each type  $A$  defined in the empty clock context  $\emptyset$ , the ‘weakened’ type in the clock context  $\kappa$  is written  $\uparrow A$ , and if we weaken all types in a context  $\Gamma$  like this, we get  $\uparrow\Gamma$ .

- (c) There are two types for streams of natural numbers in this system, the *guarded* streams  $\mathbf{Str}^g$  and the *completed* streams  $\mathbf{Str}$ . Give the definitions of these types, by replacing the dots in the definitions of  $\mathbb{N}$  and  $\mathbf{Str}^g$  by something sensible:

$$\begin{aligned} \mathbb{N} &:= \mu a. \dots \\ \mathbf{Str}^g &:= \mu a. \dots \\ \mathbf{Str} &:= \square \mathbf{Str}^g \end{aligned}$$

In other words: show that you understand how  $\mu$  is used to define inductive and coinductive types in this system. Keep in mind that  $\mathbf{Str}^g$  is a stream of natural numbers, so you will have to use  $\mathbb{N}$  in its definition. As for the clock contexts of these three types: the first and third are defined in the empty clock context  $\emptyset$ , while of course the second type  $\mathbf{Str}^g$  needs the clock context  $\kappa$ .

(d) In this system we can define functions  $\text{hd}^g$ ,  $\text{tl}^g$  and  $\text{cons}^g$ , with types:

$$\begin{aligned}\text{cons}^g &: \uparrow\mathbb{N} \rightarrow \triangleright\text{Str}^g \rightarrow \text{Str}^g \\ \text{hd}^g &: \text{Str}^g \rightarrow \uparrow\mathbb{N} \\ \text{tl}^g &: \text{Str}^g \rightarrow \triangleright\text{Str}^g\end{aligned}$$

Now from these functions define functions  $\text{hd}$ ,  $\text{tl}$  and  $\text{cons}$  on completed streams, with types:

$$\begin{aligned}\text{cons} &: \mathbb{N} \rightarrow \text{Str} \rightarrow \text{Str} \\ \text{hd} &: \text{Str} \rightarrow \mathbb{N} \\ \text{tl} &: \text{Str} \rightarrow \text{Str}\end{aligned}$$

(Note that you only need to define the latter three functions from the former three, you do not need to define the former three functions themselves.)

(e) Give the type of  $\text{fold}^g$ , which is a counterpart to the  $\text{fold}$  function from the earlier exercises that *can* be defined in this system. (Note that you do not need to define it, just giving the type is sufficient.)

## Typing rules of the lambda cube

In these rules the variables  $s, s_1, s_2$  and  $s_3$  range over the set of sorts  $\{*, \square\}$ .

*axiom*

$$\overline{\vdash * : \square}$$

*variable*

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

*weakening*

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

*application*

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

*abstraction*

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

*product*

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A. B : s_3} \text{ where } (s_1, s_2, s_3) \in \mathcal{R}$$

*conversion*

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

The sets of rules  $\mathcal{R}$  for the eight systems of the lambda cube are:

$\lambda \rightarrow$	$\mathcal{R} = \{(*, *, *)\}$
$\lambda P$	$\mathcal{R} = \{(*, *, *), (*, \square, \square)\}$
$\lambda 2$	$\mathcal{R} = \{(*, *, *), (\square, *, *)\}$
$\lambda P2$	$\mathcal{R} = \{(*, *, *), (*, \square, \square), (\square, *, *)\}$
$\lambda \underline{\omega}$	$\mathcal{R} = \{(*, *, *), (\square, \square, \square)\}$
$\lambda P\underline{\omega}$	$\mathcal{R} = \{(*, *, *), (*, \square, \square), (\square, \square, \square)\}$
$\lambda \omega$	$\mathcal{R} = \{(*, *, *), (\square, *, *), (\square, \square, \square)\}$
$\lambda P\omega = \lambda C$	$\mathcal{R} = \{(*, *, *), (*, \square, \square), (\square, *, *), (\square, \square, \square)\}$