Type Theory and Coq 2018-2019

11-06-2019

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

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

   $$(a \to b) \to (b \to c) \to (a \to c)$$

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

   (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:

   $$(a \to b) \to (b \to c) \to (a \to c)$$

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

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

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

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

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

$$\frac{}{\Gamma \vdash x : A} \quad \text{if } x : A \in \Gamma$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

(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.

$$(a \to b) \to ((a \to b) \to c \to a) \to c \to b$$

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:

$$\forall x.\, ((\forall y.\, p(y)) \to p(x))$$

(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.)

$$\cfrac{\cfrac{\cfrac{\cfrac{[p(x)^{H'}]}{p(x) \to p(x)}\, I[H']{\to} \qquad \cfrac{[\forall y.\, p(y)^{H}]}{p(x)}\, E\forall}{p(x)}\, E{\to}}{(\forall y.\, p(y)) \to p(x)}\, I[H]{\to}}{\forall x.\, ((\forall y.\, p(y)) \to p(x))}\, I\forall$$

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

$$\lambda x : D.\, \lambda H : (\Pi y : D.\, py).\, (\lambda H' : px.\, H')(Hx)$$

(c) Give the normal form of the term from the previous subexercise. Explain your answer.

By reducing the only beta redex $(\lambda H' : px.\, H')(Hx)$ in this term to Hx, we get the normal form:

$$\lambda x : D.\, \lambda H : (\Pi y : D.\, py).\, Hx$$

(d) Give the full λP typing judgement (i.e., including the λP context) of the term in normal form from the previous subexercise.

$$D : *,\, p : (D \to *) \vdash \big(\lambda x : D.\, \lambda H : (\Pi y : D.\, py).\, Hx\big) : \big(\Pi x : D.\, (\Pi y : D. py) \to px\big)$$

(e) What is the formula in minimal predicate logic that has as proof term:

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

$$(\forall x.\, p(x) \to q(x)) \to (\forall y.\, q(y) \to r(y)) \to (\forall z.\, p(z) \to r(z))$$

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

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

$$\forall a.\, (a \to a) \to a \to a$$

$$\dfrac{\dfrac{\dfrac{[a^y]}{a \to a}\, I[y]{\to}}{(a \to a) \to a \to a}\, I[x]{\to}}{\forall a.\, (a \to a) \to a \to a}\, I\forall$$

(b) Give the proof term in $\lambda 2$ for the proof from the previous subexercise.

$$\lambda a : *.\, \lambda x : a \to a.\, \lambda y : a.\, y$$

(c) Give a *different* proof term for the formula from the previous subexercises, where 'different' means that the β normal forms are different.

$$\lambda a : *.\, \lambda x : a \to a.\, \lambda y : a.\, xy$$

which can be η-reduced to:

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

Another solution is any other Church numeral, like the one for three:

$$\lambda a : *.\, \lambda x : a \to a.\, \lambda y : a.\, x(x(xy))$$

(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 $f \circ g$.

$$\lambda a : *.\, \lambda b : *.\, \lambda c : *.\, \lambda f : b \to c.\, \lambda g : a \to b.\, \lambda x : a.\, f(gx)$$

(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.

This question asks whether $\lambda 2$ is *impredicative*. And it is. So, yes, terms like this exist.

In fact, *any* term with a type of the form $\Pi a : *.\, \ldots$ has this property. For instance take for M the polymorphic identity function:

$$\begin{aligned} M &:= \lambda a : *.\, \lambda x : a.\, x \\ A &:= \Pi a : *.\, a \to a \end{aligned}$$

then we have:

$$\begin{aligned} M &: A \\ MA =_\beta (\lambda x : A.\, A) &: A \to A \end{aligned}$$

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 11 of this exam.

(a) Give a derivation in $\lambda{\to}$ of the judgement:

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{* : \Box}}{a : * \vdash a : *}}{a : *,\, x : a \vdash x : a} \qquad \dfrac{\dfrac{\dfrac{}{* : \Box}}{a : * \vdash a : *} \quad \dfrac{\dfrac{\dfrac{}{* : \Box}}{a : * \vdash a : *} \quad \dfrac{\dfrac{}{* : \Box}}{a : * \vdash a : *}}{a : *,\, x : a \vdash a : *}}{a : * \vdash (a \to a) : *}}{a : * \vdash (\lambda x : a.\, x) : (a \to a)}$$

(b) Conjunction can be impredicatively defined as:

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

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

For the lambdas, we need the rule $(\Box, \Box, \Box)$, because the type of the term is $* \to * \to *$. For the dependent product, we need the rule $(\Box, *, *)$. And for the arrows, we need the rule $(*, *, *)$. Hence we need:

$$\mathcal{R} \subseteq \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\}$$

For this reason this term can only be typed in the systems $\lambda\omega$ and $\lambda P\omega$. Other names for these systems are system F_ω and the calculus of constructions.

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

If $\Gamma \vdash M : A$, and $M \to_\beta M'$ then $\Gamma \vdash M' : A$.

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

There is no infinite reduction sequence:

$$M_0 \to_\beta M_1 \to_\beta M_2 \to_\beta \ldots$$

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
```

```
Inductive tree : Set :=
| Leaf : tree
| Node : tree -> tree -> tree.
```

(b) Give the type of the recursion principle `tree_rec` for the inductive type from the previous subexercise.

```
forall A : tree -> Set,
  A Leaf ->
  (forall t1 : tree, A t1 -> forall t2 : tree, A t2 ->
    A (Node t1 t2)) ->
  forall t : tree, A t
```

This is the dependent recursion principle, which is called `tree_rec` in Coq. The non-dependent recursion principle is also an acceptable answer to this exercise:

```
forall A : Set,
  A -> (tree -> A -> tree -> A -> A) -> tree -> A
```

(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`.

```
Fixpoint count_leaves (t : tree) {struct t} : nat :=
  match t with
  | Leaf => S O
  | Node t1 t2 => plus (count_leaves t1) (count_leaves t2)
  end.
```

(d) Define the same function using `tree_rec`.

```
tree_rec (fun _ => nat) (S O) (fun _ n1 _ n2 => plus n1 n2)
```

(e) Define an inductive predicate

$$\texttt{mirrors} : \texttt{tree} \to \texttt{tree} \to \texttt{Prop}$$

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))
```

```
Inductive mirrors : tree -> tree -> Prop :=
| mirrors_leaf : mirrors Leaf Leaf
| mirrors_node : forall t1 t1' t2 t2' : tree,
    mirrors t1 t1' -> mirrors t2 t2' ->
      mirrors (Node t1 t2) (Node t2' t1').
```

(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.)

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  eq_refl : eq A x x
```

Or, alternatively:

```
Inductive eq (A : Type) : A -> A -> Prop :=
  eq_refl : forall x : A, eq A x x
```

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.

This definition does not satisfy Coq's guard conditions. But the problem is not just an artifact of the way Coq checks productivity, the problem is that this function is just not productive for some `f`.

For example when `f` is the function that returns its second argument, the function `fold f` is not well-defined, as the definition then will not say anything about the output stream.

(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.

In Coq every function is total, which corresponds to the fact that in Coq all reductions have to terminate. The fact that all functions are total is essential for the Curry-Howard isomorphism. For this reason, type theories that are not normalizing generally are inconsistent (everything becomes provable).

In Haskell, computations do not need to terminate, which means that a function defined in Haskell can be *partial*. And therefore this `fold` function is just a partial function, and there is no problem. If one runs it on a computer with a 'bad' `f`, there just will be no output forthcoming, but there will not be any inconsistency.

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:

$$\begin{aligned}
A &::= a \mid A \to A \mid \mathbb{1} \mid A + A \mid A \times A \mid \mu a.A \mid \triangleright A \mid \Box A \mid \uparrow\! A \\
M &::= x \mid \lambda x.M \mid M\,M \mid \\
&\qquad \star \mid \mathsf{inl}\,M \mid \mathsf{inr}\,M \mid \mathsf{case}\,M\,\mathsf{of}\,x.M; x.M \mid (M, M) \mid \mathsf{fst}\,A \mid \mathsf{snd}\,A \\
&\qquad \mathsf{construct}_{\mu a.A}\,M \mid \mathsf{primrec}_{\mu a.A}\,M \mid \mathsf{fix}\,M \mid \\
&\qquad \mathsf{next}\,M \mid M \circledast M \mid \mathsf{box}\,M \mid \mathsf{unbox}\,M \mid \mathsf{force}\,M \mid \mathsf{up}\,M \mid \mathsf{down}\,M \\
\Delta &::= \varnothing \mid \kappa
\end{aligned}$$

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:

$$\frac{\Gamma \vdash_\kappa A\ \mathsf{type}}{\Gamma \vdash_\kappa \triangleright A\ \mathsf{type}} \qquad \frac{\uparrow\!\Gamma \vdash_\kappa A\ \mathsf{type}}{\Gamma \vdash_\varnothing \Box A\ \mathsf{type}} \qquad \frac{\Gamma \vdash_\varnothing A\ \mathsf{type}}{\uparrow\!\Gamma \vdash_\kappa \uparrow\! A\ \mathsf{type}}$$

And for the terms:

$$\frac{\uparrow\!\Gamma \vdash_\kappa M : A}{\Gamma \vdash_\varnothing \mathsf{box}\,M : \Box A} \qquad \frac{\Gamma \vdash_\varnothing M : \Box A}{\uparrow\!\Gamma \vdash_\kappa \mathsf{unbox}\,M : A}$$

$$\frac{\Gamma \vdash_\kappa M : A}{\Gamma \vdash_\kappa \mathsf{next}\,M : \triangleright A} \qquad \frac{\Gamma \vdash_\varnothing M : \Box \triangleright A}{\Gamma \vdash_\varnothing \mathsf{force}\,M : \Box A}$$

$$\frac{\Gamma \vdash_\varnothing M : A}{\uparrow\!\Gamma \vdash_\kappa \mathsf{up}\,M : \uparrow\! A} \qquad \frac{\uparrow\!\Gamma \vdash_\kappa M : \uparrow\! A}{\Gamma \vdash_\varnothing \mathsf{down}\,M : A}$$

In these rules, for each type A defined in the empty clock context $\varnothing$, the 'weakened' type in the clock context κ is written $\uparrow\! A$, and if we weaken all types in a context Γ like this, we get $\uparrow\!\Gamma$.

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

$$\begin{aligned}
\mathbb{N} &:= \mu a.\ \ldots \\
\mathsf{Str}^{\mathsf{g}} &:= \mu a.\ \ldots \\
\mathsf{Str} &:= \Box \mathsf{Str}^{\mathsf{g}}
\end{aligned}$$

In other words: show that you understand how μ is used to define inductive and coinductive types in this system. Keep in mind that 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 $\varnothing$, while of course the second type Str^g needs the clock context κ.

The definitions are:

$$\begin{aligned} \mathbb{N} &:= \mu a.\, \mathbb{1} + a \\ \mathsf{Str}^\mathsf{g} &:= \mu a.\, {\uparrow}\mathbb{N} \times \triangleright a \\ \mathsf{Str} &:= \Box \mathsf{Str}^\mathsf{g} \end{aligned}$$

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

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

Now from these functions define functions hd, tl and cons on completed streams, with types:

$$\begin{aligned} \mathsf{cons} &: \mathbb{N} \to \mathsf{Str} \to \mathsf{Str} \\ \mathsf{hd} &: \mathsf{Str} \to \mathbb{N} \\ \mathsf{tl} &: \mathsf{Str} \to \mathsf{Str} \end{aligned}$$

(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.)

The definitions are:

$$\begin{aligned} \mathsf{cons}\, x\, s &= \mathsf{box}\, (\mathsf{cons}^\mathsf{g}\, (\mathsf{up}\, x)\, (\mathsf{next}\, (\mathsf{unbox}\, s))) \\ \mathsf{hd}\, s &= \mathsf{down}\, (\mathsf{hd}^\mathsf{g}\, (\mathsf{unbox}\, s)) \\ \mathsf{tl}\, s &= \mathsf{force}\, (\mathsf{box}\, (\mathsf{tl}^\mathsf{g}\, (\mathsf{unbox}\, s))) \end{aligned}$$

For completeness (not required by the exercise) here are the definitions of the former three functions as well:

$$\begin{aligned} \mathsf{cons}^\mathsf{g}\, x\, s &= \mathsf{construct}_{\mathsf{Str}^\mathsf{g}}\, (x, s) \\ \mathsf{hd}^\mathsf{g}\, s &= \mathsf{primrec}_{\mathsf{Str}^\mathsf{g}}\, (\lambda z.\, \mathsf{fst}\, z) \\ \mathsf{tl}^\mathsf{g}\, s &= \mathsf{primrec}_{\mathsf{Str}^\mathsf{g}}\, (\lambda z.\, \mathsf{next}\, (\lambda x. \mathsf{fst}\, x) \circledast (\mathsf{snd}\, z)) \end{aligned}$$

(e) Give the type of $\mathsf{fold}^{\mathsf{g}}$, which is a counterpart to the `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.)

To 'solve' the productivity problem of this function, one just needs to add a single '$\triangleright$' to the type of the first argument:

$$\begin{aligned}
&\mathsf{fold}^{\mathsf{g}} : (\uparrow\mathbb{N} \to \triangleright\mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}}) \to \mathsf{Str}^{\mathsf{g}} \to \mathsf{Str}^{\mathsf{g}} \\
&\mathsf{fold}^{\mathsf{g}} = \lambda f.\, \mathsf{fix}\,(\lambda F.\, \lambda s.\, f\,(\mathsf{hd}^{\mathsf{g}}\, s)(F \circledast (\mathsf{tl}^{\mathsf{g}}\, s)))
\end{aligned}$$

Again, the definition is just given for completeness.