Type Theory and Coq 2015-2016

29-06-2016

1. (a) Consider the three untyped lambda terms:

$$I := \lambda x.\, x$$
$$K := \lambda x.\, \lambda y.\, x$$
$$S := \lambda x.\, \lambda y.\, \lambda z.\, xz(yz)$$

For each of these three terms give a most general type in the Curry-style simply typed lambda calculus.

$$I : a \to a$$
$$K : a \to b \to a$$
$$S : (a \to b \to c) \to (a \to b) \to a \to c$$

(b) Give the three terms of the Church-style simply typed lambda calculus that correspond to the typings in the previous subexercise. (I.e., give versions of these terms where types for the variables are given explicitly.)

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

(c) Give a Church-style typed lambda term for the term

$$I\, I$$

where I is the lambda term given above.

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

(d) Give a full type derivation in the simply typed lambda calculus for the term from the previous subexercise.

$$\dfrac{\dfrac{\overline{x : a \to a \vdash x : a \to a}}{\vdash (\lambda x : a \to a.\, x) : (a \to a) \to a \to a} \qquad \dfrac{\overline{y : a \vdash y : a}}{\vdash (\lambda y : a.\, y) : a \to a}}{\vdash (\lambda x : a \to a.\, x)(\lambda y : a.\, y) : a \to a}$$

(e) Give the natural deduction proof that corresponds to the lambda term in the previous two subexercises according to the Curry-Howard isomorphism.

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

(f) Does the proof from the previous subexercise contain a detour? Explain your answer. If so, also give the normal form of this proof.

Yes, there is the detour: the elimination $E{\to}$ directly follows the introduction $I[x]{\to}$. To eliminate the detour, we substitute the the proof of the right branch in the left branch, and we get:

$$\dfrac{[a^y]}{a \to a}\, I[y]{\to}$$

This proof is in normal form.

This normalization corresponds to the reduction of the proof term:

$$(\lambda x : a \to a.\, x)(\lambda y : a.\, y) \to_\beta (\lambda y : a.\, y)$$

2. (a) Give a natural deduction proof of the propositional formula

$$a \wedge b \to b \wedge a$$

$$\dfrac{\dfrac{\dfrac{[a \wedge b^x]}{b}\, Er\wedge \qquad \dfrac{[a \wedge b^x]}{a}\, El\wedge}{b \wedge a}\, I\wedge}{a \wedge b \to b \wedge a}\, I[x]{\to}$$

(b) Give the proof term for this proof according to the Curry-Howard isomorphism. You can use in this term the three functions:

$$\begin{aligned} \mathsf{conj} &: \Pi a : *.\, \Pi b : *.\, a \to b \to (a \wedge b) \\ \mathsf{proj}_1 &: \Pi a : *.\, \Pi b : *.\, (a \wedge b) \to a \\ \mathsf{proj}_2 &: \Pi a : *.\, \Pi b : *.\, (a \wedge b) \to b \end{aligned}$$

$$\lambda x : a \wedge b.\, \mathsf{conj}\; b\, a\, (\mathsf{proj}_2\; a\, b\, x)\, (\mathsf{proj}_1\; a\, b\, x)$$

(c) Give definitions of proj_1 and proj_2 using the recursor of the conjuction:

$$\mathsf{and\_ind} : \Pi a : *.\, \Pi b : *.\, \Pi c : *.\, (a \to b \to c) \to (a \wedge b) \to c$$

$$\mathsf{proj}_1 := \lambda a : *.\, \lambda b : *.\, \lambda z : a \wedge b.\, \mathsf{and\_ind}\; a\, b\, a\, (\lambda x : a.\, \lambda y : b.\, x)\, z$$
$$\mathsf{proj}_2 := \lambda a : *.\, \lambda b : *.\, \lambda z : a \wedge b.\, \mathsf{and\_ind}\; a\, b\, b\, (\lambda x : a.\, \lambda y : b.\, y)\, z$$

Or, with eta-reduction:

$$\mathsf{proj}_1 := \lambda a : *.\, \lambda b : *.\, \mathsf{and\_ind}\; a\, b\, a\, (\lambda x : a.\, \lambda y : b.\, x)$$
$$\mathsf{proj}_2 := \lambda a : *.\, \lambda b : *.\, \mathsf{and\_ind}\; a\, b\, b\, (\lambda x : a.\, \lambda y : b.\, y)$$

3. (a) Give a natural deduction proof in minimal predicate logic of:

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

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

(b) Give the proof term that corresponds to the proof from the previous subexercise under the Curry-Howard isomorphism.

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

(c) Give the full λP judgement (including the context) that gives the typing of the term from the previous subexercise. (Note that you do not need to give the *derivation* of that judgment.)

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

4. In this exercise we work in the context:

$$\Gamma := \mathsf{nat} : *,\ \mathsf{O} : \mathsf{nat},\ \mathsf{S} : \mathsf{nat} \to \mathsf{nat}$$

(a) One can encounter the following three expressions:

$$M_1 := \lambda x : \mathsf{nat}.\, \mathsf{nat}$$
$$M_2 := \Pi x : \mathsf{nat}.\, \mathsf{nat}$$
$$M_3 := \forall x : \mathsf{nat}.\, \mathsf{nat}$$

Explain what each of these expressions mean.

- M_1 is the function that maps each natural number to the type of natural numbers.
- M_2 and M_3 are two notations for the same term. This is the type of functions from natural numbers to natural numbers, $\mathsf{nat} \to \mathsf{nat}$.

(b) What are the types of these three expressions M_1, M_2 and M_3?

$$M_1 : \mathsf{nat} \to *$$
$$M_2 : *$$
$$M_3 : *$$

(c) For each of the expressions M_1, M_2 and M_3, give a term, where the expression occurs as a *proper* subterm. These terms should be well typed in the context Γ of this exercise.

$$M_1\, \mathsf{O}$$
$$\lambda f : M_2.\, f\, \mathsf{O}$$
$$\lambda f : M_3.\, f\, \mathsf{O}$$

5. (a) Is the following expression well-typed in the calculus of constructions λC?

$$(\lambda x : *.\, x)(\Pi y : *.\, y)$$

This one is subtle. According to the rules in the test, this is okay. But in Coq it depends. If you represent $*$ by Prop it is okay, but if you represent it by Set it is not okay.

(b) If your answer to the previous subexercise was 'yes', give the type of this expression. If it was 'no', explain why this is not well-typed.

$$(\lambda x : *.\, x)(\Pi y : *.\, y) : *$$

(c) Give the full λC derivation of the type judgement

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

You can find the rules of λC on page 10 of this test.

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

6. (a) Give the Coq definition of an inductive type `tree` for binary trees where the leaves do not have a label, but where the nodes are both labeled with a natural number and with a color (red or black). If you like, you can use the Coq type bool for the color, but you can also define a Coq type color for yourself, if you prefer that.

```
Definition color : Set := bool.
Definition black : color := true.
Definition red : color := false.

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

(b) Give the Coq type of the induction principle of the type you have just defined.

```
forall P : tree -> Prop,
P Leaf ->
(forall (n : nat) (c : color) (t0 t1 : tree),
 P t0 -> P t1 -> P (Node n c t0 t1)) ->
forall t : tree, P t
```

(c) Give the Coq definition of a recursive function count_nodes that counts the number of nodes in the tree. (The function that adds two natural numbers is called `plus`.)

```
Fixpoint count_nodes (t : tree) {struct t} : nat :=
  match t with
  | Leaf => O
  | Node n c t0 t1 =>
      S (plus (count_nodes t0) (count_nodes t1))
  end.
```

(d) Give the Coq definition of a predicate not_red_root that says that a tree does not have a red root (where the leaves are taken to be black).

```
Inductive not_red_root : tree -> Prop :=
| not_red_root_leaf : not_red_root Leaf
| not_red_root_node : forall (n : nat) (t0 t1 : tree),
    not_red_root (Node n black t0 t1).
```

or

```
Fixpoint not_red_root (t : tree) {struct t} : Prop :=
  match t with
  | Leaf => True
  | Node n c t0 t1 => c = black
  end.
```

or

```
Definition not_red_root (t : tree) : Prop :=
  t = Leaf
  exists n : nat, exists t1 : tree, exists t2 : tree,
    t = Node n black t1 t2.
```

(e) Give the Coq definition of an inductive predicate okay that says that in a tree of the type that you just defined, a red node will never have a red child.

```
Inductive okay : tree -> Prop :=
| okay_leaf : okay Leaf
| okay_black : forall (n : nat) (t0 t1 : tree),
```

```
    okay t0 -> okay t1 ->
    okay (Node n black t0 t1)
| okay_red : forall (n : nat) (t0 t1 : tree),
    okay t0 -> okay t1 ->
    not_red_root t0 -> not_red_root t1 ->
    okay (Node n red t0 t1).
```

7. We want a Coq formalization of the semantics of a very small imp-like language. The syntax of this language will be:

$$a ::= n \mid x \mid (a_1 \dot{-} a_2)$$
$$c ::= \mathsf{skip} \mid (x := a) \mid (c_1; c_2) \mid (\mathsf{while}\ a\ \mathsf{do}\ c\ \mathsf{od})$$

We will interpret the arithmetic expressions a as natural numbers, where subtraction is 'cut-off' subtraction (this is zero if the result would have been negative, so $4 \dot{-} 3 = 1$, but $3 \dot{-} 4 = 0$), and we will interpret the condition of the while as 'true' if the number is not equal to zero and 'false' if it is equal to zero. For convenience we also will use natural numbers as the identifiers for the variables x.

(a) Write Coq definitions of the syntax of this language as inductive types. Call the types that you define id (for the identifiers), aexp and com.

```
Definition id : Set := nat.

Inductive aexp : Set :=
| ANum : nat -> aexp
| AId : id -> aexp
| AMinus : aexp -> aexp -> aexp.

Inductive com : Set :=
| CSkip : com
| CAss : id -> aexp -> com
| CSeq : com -> com -> com
| CWhile : aexp -> com -> com.
```

(b) Write a Coq definition for a type that represents the states of this language. Call this type state.

```
Definition state : Set := id -> nat.
```

(c) Write a Coq definition for the evaluation function aeval that corresponds to $[\![a]\!]_s$. The cut-off subtraction function in Coq is called minus.

```
Fixpoint aeval (a : aexp) (s : state) {struct a} : nat :=
  match a with
  | ANum n => n
  | AId x => s x
  | AMinus a1 a2 => minus (aeval a1 s) (aeval a2 s)
  end.
```

(d) The rules of a big step semantics for this language are:

$$\frac{[\![a]\!]_s = n}{(x := a, s) \Downarrow s[x \mapsto n]}$$

$$\frac{(c_1, s) \Downarrow s' \quad (c_2, s') \Downarrow s''}{(c_1; c_2, s) \Downarrow s''}$$

$$\frac{[\![a]\!]_s = 0}{(\mathsf{while}\ a\ \mathsf{do}\ c\ \mathsf{od}, s) \Downarrow s}$$

$$\frac{[\![a]\!]_s \neq 0 \quad (c, s) \Downarrow s' \quad (\mathsf{while}\ a\ \mathsf{do}\ c\ \mathsf{od}, s') \Downarrow s''}{(\mathsf{while}\ a\ \mathsf{do}\ c\ \mathsf{od}, s) \Downarrow s''}$$

Formalize these rules as an inductive relation in Coq. Call the relation

$$\mathsf{ceval} : \mathsf{com} \rightarrow \mathsf{state} \rightarrow \mathsf{state} \rightarrow \mathsf{Prop}$$

You can use a function

$$\mathsf{update} : \mathsf{state} \rightarrow \mathsf{id} \rightarrow \mathsf{nat} \rightarrow \mathsf{state}$$

for $s[x \mapsto n]$ without defining it.

```
Inductive ceval : com -> state -> state -> Prop :=
| E_Skip : forall s, ceval CSkip s s
| E_Ass : forall x a s n, aeval a s = n ->
    ceval (CAss x a) s (update s x n)
| E_Seq : forall c1 c2 s s' s'',
```

```
    ceval c1 s s' -> ceval c2 s' s'' ->
    ceval (CSeq c1 c2) s s''
| E_WhileLoop : forall a c s s' s'',
    ~(aeval a s = 0) ->
    ceval c s s' -> ceval (CWhile a c) s' s'' ->
    ceval (CWhile a c) s s''
| E_WhileEnd : forall a c s,
    aeval a s = 0 ->
    ceval (CWhile a c) s s.
```

(e) Someone extended this until she also had a small step semantics of this language formalized in Coq, as a relation:

$$\mathsf{cstep} : (\mathsf{com} \times \mathsf{state}) \to (\mathsf{com} \times \mathsf{state}) \to \mathsf{Prop}$$

Give the Coq *statement* that says that the semantics given by ceval and the semantics given by cstep correspond to each other. You can use the function

$$\mathsf{star} : \Pi X : \mathsf{Set}. \, (X \to X \to \mathsf{Prop}) \to (X \to X \to \mathsf{Prop})$$

that gives the reflexive and transitive closure of a relation, without defining it.

```
forall (c : com) (s s' : state),
ceval c s s' <-> star (com * state) cstep (c, s) (CSkip, s')
```

8. The proof of strong normalization of the simply typed lambda calculus that was presented in the course associates a set of untyped lambda terms $[\![A]\!]$ to each simple type A. These sets are called *saturated* sets and are defined in a way that they have the two key properties:

- Each lambda term that can be typed (in the style of Curry) with type A will be in $[\![A]\!]$.
- Each term in $[\![A]\!]$ will be strongly normalizing.

Now answer the following questions:

(a) The recursive definition of $[\![A]\!]$, where A is a type of the simply typed lambda calculus, has the structure:

$$[\![a]\!] := \mathsf{SN} \qquad\qquad \text{for } a \text{ an atomic type}$$
$$[\![A \to B]\!] := \dots$$

Here SN is the set of all strongly normalizing untyped lambda terms. Complete this definition by filling in the dots for the second case.

$$[\![A \to B]\!] := \{M \mid \forall N \in [\![A]\!].\, MN \in [\![B]\!]\}$$

(b) Prove with simultaneous induction that

- $[\![A]\!] \subseteq \mathsf{SN}$
- $xN_1 \dots N_k \in [\![A]\!]$ when $N_1, \dots, N_k \in \mathsf{SN}$

(If you do not know the answer to the previous subexercise, at least prove the base case.)

- The base case is simple.
  - The first property holds by definition.
  - For the second, it is clear that a term of the shape $xN_1 \dots N_k$ with $N_1, \dots, N_k$ strongly normalizing can only reduce inside the N_i, and hence is in $\mathsf{SN} = [\![A]\!]$.
- The induction case is not much harder.
  - First we will show that $[\![A \to B]\!] \subseteq \mathsf{SN}$. Suppose that we have M with $MN \in [\![B]\!]$ for all $N \in [\![A]\!]$. We need to show that M is strongly normalizing. By induction x is in $[\![A]\!]$ (the second property with $k = 0$), so we know that $Mx \in [\![B]\!]$, and by induction that means that Mx is strongly normalizing. But if M has an infinite reduction, that also holds for Mx, so strong normalization of M follows.
  - Now to prove that $xN_1 \dots N_k \in [\![A \to B]\!]$, we need to show that $xN_1 \dots N_k N \in [\![B]\!]$ for all $N \in [\![A]\!]$. But by induction we know that these N are all strongly normalizing, so from the induction hypothesis about B the required property will follow.