Type Theory and Coq 2023-2024

17-11-2023
15:30-17:30

1. Consider the lambda term:

$$\lambda xy.\, x(y(yx))$$

Apply the PT algorithm to this term, and determine either a principal type for it, or the fact that this term is not typable in simple type theory. Explicitly give all stages of the algorithm.

We annotate all variables and applicative subterms with type variables:

$$\lambda \overset{a}{x}\overset{b}{y}.\ \underbrace{\overset{a}{x}(\underbrace{\overset{b}{y}(\underbrace{\overset{b}{y}\overset{a}{x}}_{e})}_{d})}_{c} \ :\ a \to b \to c$$

This gives rise to the following equations, which we simplify according to the PT algorithm:

$$\left\{\begin{array}{rcl} a & = & d \to c \\ b & = & e \to d \\ b & = & a \to e \end{array}\right. \overset{\text{(I)}}{\Longleftrightarrow} \left\{\begin{array}{rcl} a & = & d \to c \\ b & = & e \to d \\ b & = & (d \to c) \to e \end{array}\right. \overset{\text{(I)}}{\Longleftrightarrow}$$

$$\left\{\begin{array}{rcl} a & = & d \to c \\ b & = & e \to d \\ e \to d & = & (d \to c) \to e \end{array}\right. \overset{\text{(II)}}{\Longleftrightarrow} \left\{\begin{array}{rcl} a & = & d \to c \\ b & = & e \to d \\ e & = & d \to c \\ d & = & e \end{array}\right. \overset{\text{(I)}}{\Longleftrightarrow}$$

$$\left\{\begin{array}{rcl} a & = & d \to c \\ b & = & e \to d \\ e & = & d \to c \\ d & = & d \to c \end{array}\right. \overset{\text{(III)}}{\Longleftrightarrow} \ \mathsf{FAIL}$$

This means that the term is not typable in simple type theory.

2. Consider the following type of simple type theory:

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

Now answer the following two questions:

   (a) Give an inhabitant of this type.

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

(b) Give a type derivation that shows that this inhabitant has the appropriate type.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : a \to a \to b} \quad \dfrac{}{\Gamma \vdash y : a}}{\Gamma \vdash xy : a \to b} \qquad \dfrac{}{\Gamma \vdash y : a}}{\Gamma \vdash xyy : b}}{x : a \to a \to b \vdash (\lambda y : a.\, xyy) : a \to b}}{\vdash (\lambda x : a \to a \to b.\, \lambda y : a.\, xyy) : (a \to a \to b) \to a \to b}$$

3. Consider the following term of λP:

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

In this term there are free variables

$$\begin{aligned} D &: * \\ R &: D \to D \to * \end{aligned}$$

which represent the domain of quantification, and a binary relation on this domain.

Now answer the following three questions:

(a) This term is a proof term of a proof in predicate logic. Give the statement that is proved as a formula of predicate logic.

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

(b) Give the natural deduction proof of this statement that corresponds to this term. If you need to check a variable condition, indicate where this is the case, and why it holds.

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

For the $I\forall$ rule we need to check the variable condition that x is not free in any open assumption. The only open assumption at that point is $\forall x.\, \forall y.\, R(x, y)$, and although it does contain the variable x, it is not free in this.

(c) Does this proof contain a detour? Explain your answer.

No, it does not. There is no introduction rule that is directly followed by an elimination rule for the same connective. The proof term also does not contain a beta redex.

Note that you should not give a type derivation of this term.

4. Give type derivations in λP of the following judgment:

$$a : *,\ x : a,\ y : a \vdash x : a$$

For the rules of λP see page 7.

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

5. An impredicative definition of the natural numbers in $\lambda 2$ is:

$$\mathsf{nat}_2 := \Pi a : *.\, (a \to a) \to a \to a$$

Each natural number n is encoded as a Church numeral, which takes a type, a function and an argument, and applies that function n times to the argument. So we have that

$$n\, a\, f\, x \twoheadrightarrow_\beta f^n x$$

For example, the number three is encoded as:

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

Note that this term indeed has type nat_2.

Define a function

$$\mathsf{succ}_2 : \mathsf{nat}_2 \to \mathsf{nat}_2$$

that corresponds to the successor function on these natural numbers.

There are two natural solutions to this exercise, depending whether you add an extra function application on the outside or on the inside:

$$\mathsf{succ}_2 := \lambda n : \mathsf{nat}_2.\, \lambda a : *.\, \lambda f : a \to a.\, \lambda x : a.\, f\,(n\, a\, f\, x)$$

and:

$$\mathsf{succ}_2 := \lambda n : \mathsf{nat}_2.\, \lambda a : *.\, \lambda f : a \to a.\, \lambda x : a.\, n\, a\, f\,(fx)$$

6. We define an inductive type in Coq of binary trees with natural numbers at the leaves:

```
Inductive nattree : Set :=
| node : nattree -> nattree -> nattree
| leaf : nat -> nattree.
```

The recursor of this type has type:

```
nattree_rec
     : forall P : nattree -> Set,
       (forall t1 : nattree, P t1 ->
        forall t2 : nattree, P t2 ->
          P (node t1 t2)) ->
       (forall n : nat, P (leaf n)) ->
       forall t : nattree, P t
```

Now answer the following two questions:

(a) Define a function `nattree_sum` that adds all the natural numbers at the leaves of a tree, using `Fixpoint` and `match`. This function should have type:

```
nattree_sum
     : nattree -> nat
```

In your definition you can use the addition function on natural numbers:

```
plus
     : nat -> nat -> nat
```

```
Fixpoint nattree_sum (t : nattree) {struct t} : nat :=
  match t with
  | node t1 t2 => plus (nattree_sum t1) (nattree_sum t2)
  | leaf n => n
  end.
```

(b) Define the function `nattree_sum_rec` that computes the same sum, by applying `nattree_rec` to appropriate arguments.

```
Definition nattree_sum_rec : nattree -> nat :=
  nattree_rec (fun _ => nat)
    (fun _ n1 _ n2 => plus n1 n2) (fun n => n).
```

7. We define an inductive type in Coq of polymorphic binary trees:

```
Inductive polytree (A : Set) : Set :=
| polynode : polytree A -> polytree A -> polytree A
| polyleaf : A -> polytree A.
```

Give the type of the (dependent) induction principle `polytree_ind` for this type.

```
polytree_ind
     : forall (A : Set) (P : polytree A -> Prop),
       (forall t1 : polytree A, P t1 ->
        forall t2 : polytree A, P t2 ->
          P (polynode A t1 t2)) ->
       (forall x : A, P (polyleaf A x)) ->
       forall t : polytree A, P t
```

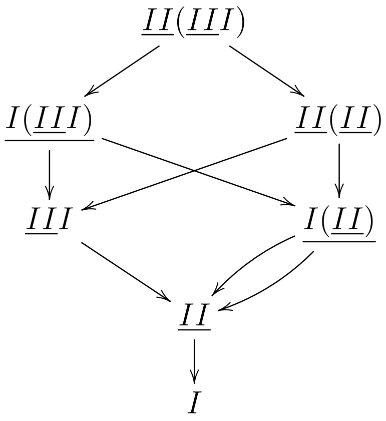
8. Consider the untyped lambda term:

$$M_8 := II(III)$$

In this we have as always that $I := (\lambda x.\, x)$.

Now answer the following two questions:

(a) Give the reduction graph of this term. If there are multiple ways to reduce a term to some other term, indicate this with multiple arrows.


$\underline{II}(\underline{II}I)$
$\underline{I(\underline{II}I)}$
$\underline{II}(\underline{II})$
$\underline{II}I$
$\underline{I(\underline{II})}$
$\underline{II}$
I

In each term we have underlined the redexes.

(b) Compute M_8^* and $(M_8^*)^*$. Just giving the answers is enough, you do not need to explain how you obtained them.

We have:

$$\begin{aligned} I^* &= I \\ (II)^* &= ((\lambda x.x)I)^* = I^* = I \\ (III)^* &= (II)^* I^* = II \end{aligned}$$

and therefore:

$$\begin{aligned} M_8^* &= (II)^*(III)^* = I(II) \\ (M_8^*)^* &= (I(II))^* = ((\lambda x.x)(II))^* = (II)^* = I \end{aligned}$$

The definition of M^* is:

$$\begin{aligned} x^* &= x \\ (\lambda x.\, M)^* &= \lambda x.\, M^* \\ (MN)^* &= \begin{cases} P^*[x := N^*] & \text{if } M = \lambda x.\, P \\ M^* N^* & \text{otherwise} \end{cases} \end{aligned}$$

9. Consider the lambda term:

$$\lambda x.\, (\lambda f.\, x)\, ((\lambda g.\, g)\, ((\lambda yz.\, y)\, x))$$

This term is typable in simple type theory. A typed version is:

$$\lambda x : a.\, (\lambda f : b \to a.\, x)\, ((\lambda g : b \to a.\, g)\, ((\lambda y : a.\, \lambda z : b.\, y)\, x))$$

or in Coq notation:

```
fun x : a => (fun f : b -> a => x)
  ((fun g : b -> a => g) ((fun (y : a) (z : b) => y) x))
```

Now answer the following three questions:

(a) Indicate what the redexes in this term are. You can do this either in the untyped or in the typed version of the term.

$$\lambda x.\, \underbrace{(\lambda f.\, x)\, (\underbrace{(\lambda g.\, g)\, (\underbrace{(\lambda yz.\, y)\, x})})}$$

The redexes in the typed term are in the corresponding places.

(b) For each of these redexes give its height.

The types of the lambda abstractions in the redexes of the typed term are respectively:

$\lambda f.\, x$	$(b \to a) \to a$
$\lambda g.\, g$	$(b \to a) \to b \to a$
$\lambda yz.\, y$	$a \to b \to a$

This means that the heights of the redexes are:

$(\lambda f.\, x)\, ((\lambda g.\, g)\, ((\lambda yz.\, y)\, x)))$	2
$(\lambda g.\, g)\, ((\lambda yz.\, y) x)$	2
$(\lambda yz.\, y)\, x$	1

(c) Indicate which redex or redexes may be contracted according to the reduction strategy from Turing's proof of weak normalization for simple type theory.

The maximal height is 2, so Turing's strategy is to pick a redex of height 2 that does not contain another redex of height 2. In this case the only redex that satisfies this is:

$$\lambda x.\, (\lambda f.\, x)\, (\underbrace{(\lambda g.\, g)\, ((\lambda yz.\, y)\, x)})$$

(Of course, contracting the outermost redex would in one step go to the normal form:

$$\lambda x.\, \underbrace{(\lambda f.\, x)\, ((\lambda g.\, g)\, ((\lambda yz.\, y)\, x))} \to_\beta \lambda x.\, x$$

But that would not be following Turing's reduction strategy!)

The height of a redex $(\lambda x : A.\, M)\, N$ is the height of the type of $(\lambda x : A.\, M)$. The height function h is defined on types by:

$$\begin{aligned} h(a) &= 0 && \text{for atomic types } a \\ h(A \to B) &= \max(h(A) + 1, h(B)) \end{aligned}$$

which implies that:

$$h(A_1 \to \cdots \to A_n \to a) = \max(h(A_1), \ldots, h(A_n)) + 1$$