MGS Lambda Calculus Exercises 3

Lyle Kopnicky

October 9, 2019

1. We can define the pointwise sum of two streams using the ana function defined in the slides:

$$\begin{aligned}\oplus = &\lambda\sigma.\ \lambda\tau.\ \mathsf{ana}\\ &(\lambda x.\ \mathsf{plus}\ (\mathsf{head}\ (\mathsf{fst}\ x))\ (\mathsf{head}\ (\mathsf{snd}\ x))\\ &(\lambda x.\ \langle(\mathsf{tail}\ (\mathsf{fst}\ x), (\mathsf{tail}\ (\mathsf{snd}\ x)))\rangle)\\ &\langle\sigma, \tau\rangle\end{aligned}$$

2. The stream of fibonacci numbers:

$$\mathsf{fibs} = Y\ (\lambda f.\ \overline{0} \triangleleft \overline{1} \triangleleft (f \oplus (\mathsf{tail}\ f)))$$

   or, in terms of ana:

$$\begin{aligned}\mathsf{fibs} = &\ \mathsf{ana}\ \mathsf{fst}\\ &(\lambda x.\ \langle\mathsf{snd}\ x, \mathsf{plus}\ (\mathsf{fst}\ x)\ (\mathsf{snd}\ x)\rangle)\\ &\langle\overline{0}, \overline{1}\rangle\end{aligned}$$

3. The operations and, or, and not on the type $\mathsf{Bool_o}$:

$$\begin{aligned}\mathsf{not_o} &= \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x : \mathsf{Bool_o} \to \mathsf{Bool_o}\\ \mathsf{and_o} &= \lambda b.\ \lambda c.\ \lambda x.\ \lambda y.\ b\ (c\ x\ y)\ y : \mathsf{Bool_o} \to \mathsf{Bool_o} \to \mathsf{Bool_o}\\ \mathsf{or_o} &= \lambda b.\ \lambda c.\ \lambda x.\ \lambda y.\ b\ x\ (c\ x\ y) : \mathsf{Bool_o} \to \mathsf{Bool_o} \to \mathsf{Bool_o}\end{aligned}$$

4. Define $\mathsf{isZero_o}$: First, the wrong way:

$$\mathsf{isZero_o} = \lambda n.\ n\ (\mathsf{and_o}\ \mathsf{false_o})\ \mathsf{true_o}$$

   This doesn't work, because n is supposed to take arguments of type $\mathsf{o} \to \mathsf{o}$ and o, but we are feeding it arguments of type $\mathsf{Bool_o} \to \mathsf{Bool_o}$ and $\mathsf{Bool_o}$.

So instead, we can move the $\lambda x.\ \lambda y.$ outside of the boolean operations:

$$\mathsf{isZero_o} = \lambda n.\ \lambda x.\ \lambda y.\ n\ (\lambda z.\ y)\ x$$

5. I'm not clear whether this exercise is to extend $\lambda\!\!\rightarrow$ with a new type constructor and set of typing rules, or whether we are expected to formulate $A \times B$ in terms of existing types in $\lambda\!\!\rightarrow$.

   If we try to formulate $A \times B$ in terms of existing types, we run into a problem, as type A must always match type B. This is because $A \times B$ would be represented by $(A \to B \to C) \to C$, but we must choose C to match either A or B. For both fst and snd to work, C must match both A and B, so A and B must be the same.

   So instead let's suppose we are to introduce a new type constructor, and pairing operator, to $\lambda\!\!\rightarrow$. Then the problem is trivial — we just set up the rules as defined in the problem, with the rules for fst and snd being defined as single reduction steps.

   So, I'm not sure what there is to do here.

6. You can't type predecessor in $\lambda\!\!\rightarrow$. Here's the definition from untyped lambda calculus:

   $$\mathsf{pred} = \lambda n.\ \mathsf{fst}\ (n\ (\lambda p.\ \mathsf{pair}\ (\mathsf{snd}\ p)\ (\mathsf{succ}\ (\mathsf{snd}\ p)))\ (\mathsf{pair}\ \overline{0}\ \overline{0}))$$

   If we want the result to be of type $\mathsf{Nat_o}$, then the $\overline{0}$s should be of type $\mathsf{Nat_o}$. But since n is applied to a function of type $\mathsf{Nat_o} \times \mathsf{Nat_o} \to \mathsf{Nat_o} \times \mathsf{Nat_o}$ and a value of type $\mathsf{Nat_o} \times \mathsf{Nat_o}$. So n must be of type $\mathsf{Nat}_{\mathsf{Nat_o} \times \mathsf{Nat_o}}$ — not the same type as the result.

   But wait — what if we used the same sort of transformation trick as we did with and and or?

   $$\mathsf{pred} = \lambda n.\ \lambda f.\ \lambda x.\ \mathsf{fst}\ (n\ (\lambda p.\ \mathsf{pair}\ (\mathsf{snd}\ p)\ (f\ (\mathsf{snd}\ p)))\ (\mathsf{pair}\ x\ x))$$

   That's closer, but still doesn't work. Now if x is of type o, then n must be of type $\mathsf{Nat_{o \times o}}$. Still not a match, since we want the result to be of type $\mathsf{Nat_o}$.

7. No, I don't think it's possible to convert between the types $\mathsf{Nat_o}$ and $\mathsf{Nat_{o \to o}}$. Either way, the conversion function would look like this (without types):

   $$\mathsf{conv} = \lambda n.\ \lambda f.\ \lambda x.\ n\ f\ x$$

But if we want to convert from $\mathsf{Nat}_{\mathsf{o}}$ to $\mathsf{Nat}_{\mathsf{o}\to\mathsf{o}}$, then n would have to have type $\mathsf{Nat}_{\mathsf{o}}$. But then on the right it would have to be applied at type $\mathsf{Nat}_{\mathsf{o}\to\mathsf{o}}$. It can't have both types.

If we want to convert the other way around, then the same problem occurs, but with the types switched around.