15-814 Homework 3 Solutions

October 25, 2017

Task 1 Define the function \( f \) in the language LNS from Homework 2, without using well-founded recursion. You will want to make use of pairs.

Solution: We compute for each \( n \) the pair \( \langle f(n), f(n+1) \rangle \) and take the first projection. We assume that the usual + has been defined, e.g. see Homework 2’s solution notes.

\[
 f = \lambda n : \text{nat}. \pi_1 \text{natrec}(n; \langle z, s(z) \rangle; x.y. \langle \pi_2 y, \pi_1 y + \pi_2 y \rangle)
\]

Task 2 Define the function \( f \) in System T, using well-founded recursion.

Solution: We first define the function \( F \) for which we want to perform a fixed-point computation. We again assume that ifz, \( \leq \), \(-\), \(+\) have been defined following Homework 2.

\[
 F = \lambda f : \text{nat} \rightarrow \text{nat}. \lambda n : \text{nat}. \text{ifz} (n \leq s(z)) \ n \ (f(n - s(s(z))) + f(n - s(z)))
\]

It is useful to note its type \( F : (\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat}) \). A fixed point of \( F \) would be an \( x : \text{nat} \rightarrow \text{nat} \) such that \( Fx = x \). Next, we define the n-fold composition of a function with itself.

\[
 \text{ncompose} = \lambda f : (\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat}). \lambda n : \text{nat}. \text{natrec}(n; \lambda m : \text{nat}. m; x.y.f y)
\]

Finally, we put the two together to define what we want.

\[
 f = \lambda n : \text{nat}. \text{ncompose} F n n
\]

To get a better understanding of what is going on, let us manually unfold \( f \). (I write \( \equiv \) here because some of these steps are not in correct evaluation order). Notice that composing \( F \) with itself 3 times is sufficient, because we always decrease \( n \) by at least 1 in each recursive call.

\[
 f 3 \equiv \text{ncompose} F 3 3 \\
 \equiv \text{natrec}(3; \lambda m : \text{nat}. m; x.y. F y) 3 \\
 \equiv F (F (F (\lambda m : \text{nat}. m))) 3 \\
 \equiv (F^3 \text{id}) 3 \\
 \equiv \text{ifz} (3 \leq 1) 2 ((F^2 \text{id}) (3 - 2) + (F^2 \text{id}) (3 - 1)) \\
 \equiv (F^2 \text{id}) 1 + (F^2 \text{id}) 2 \\
 \equiv (F^2 \text{id}) 1 + (F \text{id}) 0 + (F \text{id}) 1 \\
 \equiv 2
\]

Essentially, the n-fold composition acts like a “clock” or “fuel” for \( F \) that is decremented on each recursive call. This ensures well-foundedness because we cannot decrement the clock forever.
Task 3  For each of the following pairs of types $\tau_1, \tau_2$, define expressions $f, g$ with types $f : \tau_1 \rightarrow \tau_2$ and $g : \tau_2 \rightarrow \tau_1$ respectively.

Solution: I write $\lambda a : \tau, b : \tau'.e$ as shorthand for $\lambda a : \tau.\lambda b : \tau'.e$. Note that we can also understand these using the Curry-Howard isomorphism. For example, (Triple Negation) corresponds to the tautology $\neg \neg p \iff p$ that was mentioned in class.

1. (Currying) $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3), (\tau_1 \times \tau_2) \rightarrow \tau_3$
   
   \[ f = \lambda x : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3), p : \tau_1 \times \tau_2. x (\pi_1 p) (\pi_2 p) \]
   
   \[ g = \lambda x : \tau_1 \times \tau_2 \rightarrow \tau_3, \lambda p_1 : \tau_1, p_2 : \tau_2. (p_1; p_2) \]

2. (Permutation) $(\tau_1 + \tau_2) + \tau_3, (\tau_2 + \tau_3) + \tau_1$
   
   \[ f = \lambda x : (\tau_1 + \tau_2) + \tau_3. \text{case } x \{ y. \text{case } y \{ \text{inr}(l); \text{inl}(r)); y. \text{inl}(\text{inr}(y)) \} \}
   
   Similarly for $g$.

3. (Distributivity) $\tau \times (\tau_1 + \tau_2), (\tau \times \tau_1) + (\tau \times \tau_2)$
   
   \[ f = \lambda x : \tau \times (\tau_1 + \tau_2). \text{case } \pi_2 x \{ y. \text{inl}(\langle \pi_1 x, y \rangle); y. \text{inr}(\langle \pi_1 x, y \rangle) \}
   
   \[ g = \lambda x : (\tau \times \tau_1) + (\tau \times \tau_2). \text{case } x \{ y. \text{inl}(\langle \pi_2 y \rangle); y. \text{inl}(\langle \pi_2 y \rangle) \}
   
4. (De Morgan) $(\tau_1 + \tau_2) \rightarrow \text{void}, (\tau_1 \rightarrow \text{void}) \times (\tau_2 \rightarrow \text{void})$
   
   \[ f = \lambda x : (\tau_1 + \tau_2) \rightarrow \text{void}. (\lambda y : \tau_1. \text{inl}(y), \lambda y : \tau_2. \text{inr}(y)) \]
   
   \[ g = \lambda x : (\tau_1 \rightarrow \text{void}) \times (\tau_2 \rightarrow \text{void}). \text{case } y \{ z. \pi_1 x z; z. \pi_2 x z \}
   
5. (Soundness) $\tau \times (\tau \rightarrow \text{void}), \text{void}$
   
   \[ f = \lambda x : \tau \times (\tau \rightarrow \text{void}). (\pi_2 x) (\pi_1 x) \]
   
   \[ g = \lambda x : \text{void}. \text{abort}[(\tau \times (\tau \rightarrow \text{void)}](x) \]

6. (Triple Negation) $((\tau \rightarrow \text{void}) \rightarrow \text{void}) \rightarrow \text{void}, \tau \rightarrow \text{void}$
   
   \[ f = \lambda x : ((\tau \rightarrow \text{void}) \rightarrow \text{void}) \rightarrow \text{void}. y : \tau. x (\lambda v : \tau \rightarrow \text{void}. v y) \]
   
   \[ g = \lambda x : \tau \rightarrow \text{void}. y : ((\tau \rightarrow \text{void}) \rightarrow \text{void}). y x \]

Task 4  Lists can either be Nil or Cons of an element and a list. In ML-style, we write

\[ \tau \text{ list } \triangleq \text{Nil } \mid \text{Cons of } \tau \times \tau \text{ list} \]

Define the following types and functions in System F. As usual, briefly explain (in 1-2 lines) the intuition behind your answer:

1. Define $\tau \text{ list }$ in System F.

2. Define $\text{nil } : \tau \text{ list }$, the empty list.

3. Give the representation of the list $[a_1, a_2, \ldots, a_n]$ where $a_i : \tau$.

4. Define $\text{cons } : \tau \rightarrow \tau \text{ list } \rightarrow \tau \text{ list }$.  

2
5. Define listrec\((l, e_0, x.y.e_1) : \rho\) (analogous to natrec in System $T$). Its static and dynamic semantics are given below. Here, $x$ is bound to the head of the list, and $y$ is bound to the result of the computation on the tail of the list.

\[
\begin{align*}
\Gamma \vdash l : \tau \text{ list} & \quad \Gamma \vdash e_0 : \rho & \quad \Gamma, x : \tau, y : \rho \vdash e_1 : \rho \\
\hline
\Gamma \vdash \text{listrec}(l, e_0, x.y.e_1) : \rho & \quad \text{(LISTREC)}
\end{align*}
\]

\[
\text{listrec}(\text{nil}, e_0, x.y.e_1) \mapsto e_0 \quad \text{(LISTREC-NIL)}
\]

\[
\text{listrec}(\text{cons}(h, t), e_0, x.y.e_1) \mapsto [h, \text{listrec}(t, e_0, x.y.e_1)/x, y]e_1 \quad \text{(LISTREC-S)}
\]

6. Define the function append \((\tau \text{ list} \to \tau \text{ list} \to \tau \text{ list})\), which takes two lists and appends the second list at the end of the first. (You can define this by working it out with listrec, but there is a cleaner, more elegant solution.)

Solution:

1. Note the similarity between this and natural numbers (what if $\tau = \text{unit}$?)

   \[\tau \text{ list} = \forall \alpha. \alpha \to (\tau \to \alpha \to \alpha) \to \alpha\]

2.

   \[\text{nil}_\tau : \tau \text{ list} = \Lambda \alpha. \lambda m : \alpha. \lambda c : \tau \to \alpha \to \alpha.m\]

3.

   \[[a_1, a_2, \ldots, a_n] = \Lambda \alpha. \lambda m : \alpha. \lambda c : \tau \to \alpha \to \alpha.c\ a_1(c\ a_2(\ldots (c\ a_n\ m))))\]

4.

   \[\text{cons}_\tau : \tau \to \tau \text{ list} \to \tau \text{ list} = \lambda h : \tau. \lambda t : \tau \text{ list}. \Lambda \alpha. \lambda m : \alpha. \lambda c : \tau \to \alpha \to \alpha.c\ h\ (t[\alpha]\ m\ c)\]

5.

   \[\text{listrec}(l : \tau \text{ list}, e_0, x.y.e_1) : \rho = l[\rho]\ e_0\ (\lambda x : \tau. \lambda y : \rho.e_1)\]

6.

   \[\text{append} : \tau \text{ list} \to \tau \text{ list} \to \tau \text{ list} = \lambda x : \tau \text{ list}. \lambda y : \tau \text{ list}. \Lambda \alpha. \lambda m : \alpha. \lambda c : \tau \to \alpha \to \alpha.x[\alpha]\ (y[\alpha]\ m\ c)\ c\]