Homework 5: Type Theory

Consider this homework to be "due" on December 7.

1. Show that $\vdash_{\lambda\mu} (\lambda x\,.\,xx)(\lambda x\,.\,xx) : \sigma$ in the system $\lambda\mu$-Curry.

2. In the system $\lambda\cap$-Curry, show that

   (a) $\vdash_{\lambda\cap} (\lambda x\,.\,xx) : ((\sigma \cap (\sigma \to \tau)) \to \tau)$.

   (b) $\vdash_{\lambda\cap} (\lambda xyz\,.\,xz(yz))(\lambda xy\,.\,x) : (\alpha \to \beta \to \beta)$.

3. Define the system $\lambda\cap$-Church.

4. Given the following definitions in $\lambda 2$-church:

$$\begin{aligned}
\mathrm{Nat} &= \forall t\,.\,t \to (t \to t) \to t \\
0 : \mathrm{Nat} &= \Lambda t\,.\,\lambda z : t\,.\,\lambda s : t \to t\,.\,z \\
\mathrm{succ} : \mathrm{Nat} \to \mathrm{Nat} &= \lambda n : \mathrm{Nat}\,.\,\Lambda t\,.\,\lambda z : t\,.\,\lambda s : t \to t\,.\,s(n\ t\ z\ s) \\
\mathrm{It} : \forall t\,.\,(t \to t) \to \mathrm{Nat} \to t \to t &= \Lambda t\,.\,\lambda f : t \to t\,.\,\lambda n : \mathrm{Nat}\,.\,\lambda u : t\,.\,n\ t\ u\ f
\end{aligned}$$

   Show that the following properties hold (for all types T, for all $f : T \to T$, for all $u : T$):

   (a) $\mathrm{It}\ T\ f\ 0\ u = u$,

   (b) $\mathrm{It}\ T\ f\ (\mathrm{succ}\ n)\ u = f\ (\mathrm{It}\ T\ f\ n\ u)$.

5. In $\lambda P\omega$, define a constructor "list", with constant "nil" and operation "cons", and an iteration form "reduce" with the following properties for all types U (i.e., for all $U : *$), for all $e : U$, for all $l : (\mathrm{list}\ U)$:

$$(\mathrm{nil}\ U) : (\mathrm{list}\ U)$$

$$(\mathrm{cons}\ U\ e\ l) : (\mathrm{list}\ U)$$

   and in addition, for all tylpes W, for all $w : W$, for all $f : U \to W \to W$,

$$(\mathrm{reduce}\ U\ W\ w\ f\ l) : W$$

$$\begin{aligned}
(\mathrm{reduce}\ U\ W\ w\ f\ (\mathrm{nil}\ U)) &= w \\
(\mathrm{reduce}\ U\ W\ w\ f\ (\mathrm{cons}\ U\ e\ l)) &= f\ e\ ((\mathrm{reduce}\ U\ W\ w\ f\ l)).
\end{aligned}$$

   Hint: much of this can be done in $\lambda 2$-Church, but be sure to use the notation of $\lambda P\omega$.

6. This exercise introduces existential or dependent sum types. Such types are useful for modeling data abstraction, and modular structures as evidenced by the Mitchell and Harper paper "The essence of ML", sections 5–11.

   An element of a dependent sum type can be thought of as an ordered pair, where the type of the second element may depend on the *value* of the first. A trivial example is the type $\Sigma T : *\,.\,T$, with elements such as $(\mathrm{Nat}, 0)$ and $(\mathit{Boolean}, \mathit{true})$. Another example is $\Sigma n : \mathrm{Nat}\,.\,(\mathrm{MultipleOf}\ n)$, with elements such as $(3, 24)$, and $(5, 50)$. The idea is that one uses a type of the form $\Sigma T : * \,.\, \mathrm{rep}$, where rep is a record type, containing various operations on the hidden type T. For example, if one was defining a type of stacks of natural numbers, then one might use the following definition:

$$\begin{aligned}\mathrm{Stack} \;&:\; \Sigma T : * \,.\, \textbf{record}\ \mathrm{empty} : T, \mathrm{push} : T \to \mathrm{Nat} \to T, \mathrm{pop} : T \to T, \ldots\ \textbf{end} \\ &=\; (\mathrm{NatList}, \textbf{record}(\mathrm{empty} = \mathrm{nil}, \mathrm{push} = \mathrm{cons}, \mathrm{pop} = \mathrm{car}, \ldots))\end{aligned}$$

Below we will encode rules for dependent sum types that do not enforce data abstraction. (See the literature survey for references on how to enforce data abstraction with existential types.) According to Girard, in $\lambda 2$, when V is a type expression, and X is a type variable, then one can define a dependent sum type as the following syntactic sugar:

$$\Sigma X : * \,.\, V \stackrel{\mathrm{def}}{=} \Pi Y : * \,.\, (\Pi X : * \,.\, V \to Y) \to Y$$

(using the notation of $\lambda P\omega$). Note that V can have X as a free variable. In $\lambda P\omega$ we can express the dependency of V on X by abstracting out the type variable X, making a constructor

$$V' : * \to * \stackrel{\mathrm{def}}{=} \lambda X : * \,.\, V$$

We can thus interpret Σ itself as a constructor that takes a constructor of kind $* \to *$ as an argument, and returns a type:

$$\Sigma' : (* \to *) \to * = \lambda V' : (* \to *) \,.\, \Pi Y : * \,.\, (\Pi Z : * \,.\, (V'\ Z) \to Y) \to Y$$

So that we can regard $\Sigma X : * \,.\, V$ as sugar for $\Sigma'\ (\lambda X : * \,.\, V)$ (i.e., $\Sigma'\ V'$).

(a) Complete the following definitions of the operations pack and unpack which respectively construct and observe elements of dependent sum types.

   i. $\mathrm{pack} : \Pi V' : (* \to *) \,.\, \Pi U : * \,.\, (V'\ U) \to (\Sigma'\ V') = \ldots$

   ii. $\mathrm{unpack} : \Pi V' : (* \to *) . \Pi t : (\Sigma'\ V') . \Pi W : * . \Pi w : (\Pi X : * \,.\, (V'\ X) \to W) . W = \ldots$

   iii. $\mathrm{fst} : \Pi V' : (* \to *) \,.\, \Pi t : (\Sigma'\ V') \,.\, * = \ldots$

   iv. $\mathrm{snd} : \Pi V' : (* \to *) \,.\, \Pi t : (\Sigma'\ V') \,.\, V'(\mathrm{fst}\ t) = \ldots$

(b) Show that your definitions have the following properties.

   i. $(\mathrm{unpack}\ (\lambda X : * \,.\, V)\ (\mathrm{pack}\ (\lambda X : * \,.\, V)\ U\ v)\ W\ (\lambda X : * \,.\, \lambda x : (V'\ X) \,.\, w))$
   $= (\lambda X : * \,.\, \lambda x : (V'\ X) \,.\, w)\ U\ v.$

   ii. $(\mathrm{fst}\ (\lambda X : * \,.\, V)\ (\mathrm{pack}\ (\lambda X : * \,.\, V)\ U\ v)) = U.$

   iii. $(\mathrm{snd}\ (\lambda X : * \,.\, V)\ (\mathrm{pack}\ (\lambda X : * \,.\, V)\ U\ v)) = v.$