$\boxed{\text{CIS 670 Homework 1, Due Sept. 25, 2002}}$

1. Prove preservation for System F. If $\emptyset; \emptyset \vdash e : \tau$ and $e \to e'$ then $\emptyset; \emptyset \vdash e' : \tau$.

2. Prove progress for System F. If e is a closed, well-typed term, then either e is a value or else there is some e' with $e \to e'$.

3. Derive an encoding for the following ML datatype in System F. Implement it in ocaml using only function abstraction and application.

```
data Tree = Leaf | Node of int * Tree * Tree
```

4. (Pierce, TPL 23.4.9) Write a function `pred` that computes the predecessor of a Church numeral (returning 0 if its input is 0). The key idea is to define a function f that maps the pair (i, j) into the pair $(i + 1, i)$—that is, it throws away the second component of its argument, copies the first component to the second, and increments the first. Then n applications of f to the starting pair $(0, 0)$ yields the pair $(n, n - 1)$.

5. Existential types are useful in *typed compilation.* Before doing anything else, read the first two sections of Harper and Morrisett's paper "Typed-Closure Conversion for Recursively-Defined Functions". These two sections describe how to use existential types to implement typed-closure conversion for the simply-typed lambda calculus.

   Your job is to extend this translation to System F, by extending the target language with universal types. You must extend the translation (in Figure 1) so that it includes cases for type abstraction and application.

$$\begin{array}{lcl} (\text{types})\tau & ::= & \ldots \mid \forall t.\tau \\ (\text{terms})e & ::= & \ldots \mid \Lambda t.e \mid e[\tau] \end{array}$$

   In System F , a function $\lambda x : \tau.e$ may have both free term variables and free *type* variables. In the simply-typed case, a closure contains a code pointer and an environment containing the values for the free term variables. We could do something similar for the free type variables and create an analogous *type environment*, but that involves several additions to the target caluclus (see Minimide, Morrisett, Harper "Typed Closure Conversion" for the gorey details).

   However, we will use a trick invented by Morrisett et al. for Typed Assembly Language. The purpose of closure conversion is to convert code to a version that more closely resembles a language that the machine executes, for example, x86 assembly language. A later step in typed compilation is *type erasure.* Even though the operational semantics of System F passes type information at run-time, those types never have any effect. So it would be just as good if we executed an version of the program where all the types have been erased (see the end of this section for a formal definition of type erasure).

Knowing that all type information will be erased, how should we change the case ($\rightarrow$ $-\mathbf{I}$) in Figure 1 to handle the free type variables of the function? When we create a closure $\langle v_{code}, v_{env}\rangle$ we delay the application of the function v_{code} to its environment v_{env} until the function is actually called. So that v_{code} remains closed (allowing us to hoist it), we need to abstract all of the free type variables in e. So the definition of v_{code} must be wrapped by $\Lambda\alpha_1.\ldots\Lambda\alpha_n.$ where $\alpha_1\ldots\alpha_n$ are the free type variables of e.

However, when we create the closure, it is ok to eagerly apply the function v_{code} to its *type* environment, because that type instantiation does not have a compuational effect. In the end, we're going to erase this type application, so it doesn't matter if we do the application now or later when the function is called. Therefore, the second change we need to do is modify the closure to be $\langle v_{code}[\alpha_1]\ldots[\alpha_2], v_{env}\rangle$.

With this definition, because we eagerly apply the type arguments, the case for application ($\rightarrow$ $-\mathbf{E}$) does not need to be modified.

Type erasure There is still one issue remaining with type erasure that you must be careful of when you defined the cases for type abstraction and type application.

The untyped calculus looks like:

$$u ::= x \mid c \mid \texttt{code}(x.u) \mid \texttt{call}\ u_1 u_2 \mid \langle u_1, \ldots, u_n\rangle \mid \pi_i u \mid \texttt{let}\ x = u'\,\texttt{in}\ u$$

In order for this trick to work, type erasure must completely eliminate type applications and not replace them with anything else. Therefore, we will use the definition of type erasure $|e|$ shown below.

$$\begin{array}{lcl}
|x| & = & x \\
|c| & = & c \\
|\texttt{code}\ (x:\tau).u| & = & \texttt{code}\ x.|u| \\
|\texttt{call}\ e_1 e_2| & = & \texttt{call}\ |e_1||e_2| \\
|\langle e_1, \ldots, e_n\rangle| & = & \langle |e_1|, \ldots, |e_n|\rangle \\
|\pi_i e| & = & \pi_i |e| \\
|\texttt{let}\ x = e_1\,\texttt{in}\ e_2| & = & \texttt{let}\ x = |e_1|\,\texttt{in}\ |e_2| \\
|\Lambda\alpha.e| & = & |e| \\
|e[\tau]| & = & |e| \\
|\texttt{pack}[\tau, e]| & = & |e| \\
|\texttt{let}\ t, x = \texttt{unpack}\ e_1\,\texttt{in}\ e_2| & = & \texttt{let}\ x = |e_1|\,\texttt{in}\ |e_2|
\end{array}$$

This definition works if the output of closure conversion restricts what terms may be polymorphic—only values may abstract types (i.e. type abstractions must be of the form $\Lambda\alpha.v$). Otherwise, the erased language may not behave exactly the same as the original language. Therefore, we must respect this restriction in the closure-conversion translation. (See Pierce, TPL chapter 23.7 for a more robust version of type erasure.)

To maintain this property, use the following as the type translation of polymorphic types:

$$\mathcal{T}[\![\forall t.\tau]\!] = \exists t'.\langle(\forall t.t' \Rightarrow \mathcal{T}[\![\tau]\!]), t'\rangle$$