Problem Set 9
Boston, December 5th, 2006.
Michel Silva Machado (U00056504)

Problem 1. Using map as a model, write a polymorphic list-reversing function:

reverse : ∀X. List X → List X.

```
_reverse = λX.
               (fix (λrev: (List X) → (List X) → (List X).
                    λl: List X. λanswer: List X.
                         if isnil [X] l
                         then answer
                         else rev (tail [X] l)
                                  (cons [X] (head [X] l) answer)))

reverse = λX. λl: List X. _reverse [X] l (nil [X])
```

The extra parameter "answer" in "_reverse" accumulates the reversed list until the original list "l" is empty. A close implementation of this function in OCaml and some test cases are showed below.

```
(michel) ~/ % ocaml
        Objective Caml version 3.09.1

# let rec _reverse (l: 'a list) (answer: 'a list) =
  if (List.length l) = 0
  then answer
  else _reverse (List.tl l)
                ((List.hd l) :: answer);;
val _reverse : 'a list -> 'a list -> 'a list = <fun>
# let reverse (l: 'a list) = _reverse l [];;
val reverse : 'a list -> 'a list = <fun>
# reverse [];;
- : 'a list = []
# reverse [1];;
- : int list = [1]
# reverse ["a string"];;
- : string list = ["a string"]
# reverse [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]
```

Problem 2. Show that the type

PairNat = ∀X. (CNat → CNat → X) → X;

can be used to represent pairs of numbers, by writing functions

pairNat : CNat → CNat → PairNat;

fstNat : PairNat → CNat;

sndNat : PairNat → CNat;

for constructing elements of this type from pairs of numbers and for accessing their first and second components.

```
pairNat = λn1: CNat. λn2: CNat.
              λX. λfunc: CNat → CNat → X. func n1 n2
fstNat = λpair: PairNat. pair [CNat] (λn1: CNat. λn2: CNat. n1)
sndNat = λpair: PairNat. pair [CNat] (λn1: CNat. λn2: CNat. n2)
```

(Comment.) The fact that X can be any type is not explored in this question, in fact, one could change X to CNat, remove the type abstraction, and have a simpler code. However, there are cases where having a polymorphic return type is interesting. The following example assumes that records are part of the language.

```
PairNat_to_2D_point = λpair: PairNat.
     pair [{x=CNat, y=CNat}] (λn1: CNat. λn2: CNat. {x=n1, y=n2})
```

Problem 3. (1) How many different supertypes does {a: Top, b: Top} have? (2) Can you find an infinite descending chain in the subtype relation – that is, an infinite sequence of types S_0, S_1, etc. such that each S_{i+1} is a subtype of S_i? (3) What about an infinite ascending chain?

(1) The supertypes of size 2: {a: Top, b: Top} and {b: Top, a: Top}; The supertypes of size 1: {a: Top} and {b: Top}; The supertype of size 0: {}; and, finally Top. Total: 6 supertypes.

(2) S_0 = {}; S_1 = {label1: Top}; S_2 = {label1: Top, label2: Top}; ...;
S_n = {label1: Top, label2: Top, ..., labeln: Top}

(3) In order to get an ascending chain, one can not rely on finding a type T_0 whose ascending types are the previous type with something removed, the opposition to the previous adding process, because since types are finite here, it would not produce an infinite sequence. Thus, an infinite sequence must rely on adding something to the previous term. This can only happens to the arrow type because it has the subtype relation inverted to the left side of the arrow. Therefore the answer to this item is as follow: $T_0 = S_0 \rightarrow$ Top; $T_1 = S_1 \rightarrow$ Top; $T_2 = S_2 \rightarrow$ Top; ...; $T_n = S_n \rightarrow$ Top.

(Comment.) In the previous examples, “Top” behaves just as a placeholder, it can be replaced by any valid type since the subtype relation is reflexive (X <: X). The given definition of descending / ascending chains does not seem to be what Pierce really meant when he devised this question because the following sequence is valid for both item (2) and item (3): $S_0 = T_0$ = Top; $S_1 = T_1$ = Top; $S_2 = T_2$ = Top; ...; $S_3 = T_3$ = Top. Once again, Top is used just as a placeholder and the previous sequence is valid because (again) subtype relation is reflexive.

Problem 4. Prove the decidability of the type-checking problem relative to the type-annotated rules (page 4 of handout 22).

Let t be a given term of the language defined on page 2 in handout 22, and Γ a given context. Type checking t under the assumptions held in Γ and giving the type of t if it exists are decidable.

Proving by induction on the structure of terms. All the typing rules referenced below can be found on page 4 in handout 22.

Case t = x (variable)

if $(x : T) \in \Gamma$, the type of t is T, according to the rule T-Var. Otherwise, t does not type-check under Γ.

Case t = λx: T. t' (abstraction)

Using the inductive hypothesis, if t' type-checks under the assumptions $\Gamma' = \Gamma, (x : T)$, t' has type T', otherwise t' does not type-check. If t' does not type-checks, neither does t. If t' type-checks, t has type $T \rightarrow T'$ according to T-Abs.

Case t = t_1 t_2 (application)

Using the inductive hypothesis, if t_1 type-checks under the assumptions Γ, t_1 has type T_1, otherwise t_1 does not type-check. For t_2, If t_2 type-checks under the assumptions Γ, t_2 has type T_2, otherwise t_2 does not type-check. If t_1 or t_2 does not type-checks, neither does t.

If t_1 and t_2 type-check,$T_1 = T_{left} \rightarrow T_{right}$, and $T_{left} = T_2$, then t has type T_{right} according to T-App, otherwise t does not type-check.

Case t = λX. t' (type abstraction)

Using the inductive hypothesis, if t' type-checks under the assumptions $\Gamma' = \Gamma, X$, t' has type T', otherwise t' does not type-check. If t' does not type-checks, neither does t. If t' type-checks, t has type $\forall X.T'$ according to T-TAbs.

Case t = t' [T] (type application)

Using the inductive hypothesis, if t' type-checks under the assumptions Γ, t' has type T', otherwise t' does not type-check. If t' type-checks and $T' = \forall X.T''$, then the type of t is the result of the substitution $[X \rightarrow T]T''$ according to T-TApp, otherwise t does not type-check.