Homework 6

Please submit this work by Thursday, May 22 at 11:59pm.

Please TeXify your responses to 2 and 3 and submit them in a file `hw6.pdf`.

Problem 0: COQ

This is an honor-bound homework exercise with nothing to submit. (Yes, I did hear you choke on your coffee just now.) Work through the chapters*Lists* and *Poly* in *Software Foundations*.

Problem 1: SML Records

Since we are entering a unit on subtyping, and since our treatment of subtyping is concerned with records, this exercise is intended to give you experience programming with records in Standard ML, and, as a secondary effect, give you still more experience with the ML approach to program design.

The particular exercise here is to implement a form of instrumented lists. The instrumentation of these lists is a precomputed length, maximum element and minimum element, such that length, max and min are constant-time operations when called. Please be clear that an instrumented list is specifically not a sorted list, just a list that records, at every node, its length, max and min. Put differently, InstrumentedList.cons is just cons, not insert.

Your repository will have been seeded with starter code before you begin this exercise. The signature`INSTRUMENTED_LIST` specifies, through types alone, the bulk of this exercise to a great extent. (Some of the types are likely to make sense once you've started writing code, not necessarily sooner.) Nevertheless, explanatory comments throughout the signature (reproduced below) are intended to clarify.

```
signature INSTRUMENTED_LIST = sig

  type 'a ord   = {lt : 'a * 'a -> bool}
  type 'a instr = {len : int, max : 'a, min : 'a}

  datatype 'a seq
    = Nil
    | Cons of 'a instr * ('a * 'a seq)

  datatype 'a ilist
    = List of 'a ord * 'a seq

(* return NONE for empty list *)
  val hd       : 'a ilist -> 'a option

(* return NONE for empty list *)
  val tl       : 'a ilist -> 'a ilist option

(* make sure instrumentation is correctly maintained! *)
  val cons     : 'a * 'a ilist -> 'a ilist

(* the following three must all be constant-time operations *)
  val length   : 'a ilist -> int
  val max      : 'a ilist -> 'a option
  val min      : 'a ilist -> 'a option

(* the following higher-order operations must be completed in a single pass *)
(* i.e., converting to a regular list, using map/filter on that, then converting *)
(* back again, is not allowed *)
  val map      : 'b ord -> ('a -> 'b) -> 'a ilist -> 'b ilist
  val filter   : ('a -> bool) -> 'a ilist -> 'a ilist
  val foldr    : ('a * 'b -> 'b) -> 'b -> 'a ilist -> 'b
  val foldl    : ('a * 'b -> 'b) -> 'b -> 'a ilist -> 'b

(* make sure instrumentation is correctly maintained! *)
  val rev      : 'a ilist -> 'a ilist

(* for "same", since the "ord" components of two ilists can't be compared, *)
(* just compare instrumentation data and list data *)
  val same     : ('a * 'a -> bool) -> 'a ilist * 'a ilist -> bool

  val toList   : 'a ilist -> 'a list
  val fromList : 'a ord -> 'a list -> 'a ilist

end
```

Note that Harper's ML text [`http://www.cs.cmu.edu/~rwh/smlbook/book.pdf`] treats records, including record pattern matching, inChapter 5.

Problem 2: Orthogonal Language Features all Stuck Together

State the type soundness theorem and preservation and progress theorems for the simply-typed lambda calculus supplemented with unit, fix per section 11.11, references per chapter 13, and errors per figure 14.2. (You do not need to prove the theorems, just state them.)

Problem 3: Record Subtyping

Section 15.2 of *TaPL* gives the following rule for subtyping of function types.

```
T1 <: S1    S2 <: T2
--------------------    (arrow subtyping)
S1 -> S2 <: T1 -> T2
```

Along with the subsumption rule, which follows, and the usual rule for typed application, we have a way to check applications to supertypes.

```
G |- t : S   S <: T
-------------------    (subsumption)
    G |- t : T
```

The tricky part of understanding the function subtyping rule is understanding why, in the premises, the relations point the way they do (contravariant, meaning "pointing the opposite way", for arguments, and covariant, meaning "pointing the same way", for result types). The claim is that they must so point in order not to sabotage the usual soundness properties.

Assume arrow subtyping were instead as follows.

```
S1 <: T1    S2 <: T2
--------------------    (both-covariant)
S1 -> S2 <: T1 -> T2
```

Then write a well-typed program that goes wrong.

Next, do likewise for this rule.

```
T1 <: S1    T2 <: S2
```

```
     --------------------    (both-contravariant)
     S1 -> S2 <: T1 -> T2
```

You may write the programs in each case in bare plaintext abstract syntax, as the text does. You do not need to produce the relevant typing derivations as part of the homework (we will spare you the TeX hackery); just the errant programs are enough. Nevertheless, sketching those derivations on paper as you work out your answers is recommended.