Homework # 11
due May 13th, 13:00

Turn in your solution as two files `systemF.slf` and `list.f` by email to `scmalte@inf.ethz.ch`.

1 Reading

Please read Chapters 23 and 24 in your textbook.

2 Proofs

Prove the type soundness of pure System F (see Figure 23-1, page 343) in SASyLF in the same style as previous proofs (canonical forms, progress, substitution and preservation). You are not given a skeleton file, but can start with `stlc.slf`, remove booleans and add the new System F specific terms, contexts, values, evaluation forms and type rules. Up to now, we had only one binding context in the environment, now we will need another binding: Γ`, X`, and a new judgment to support it.

Use the partial solutions to Exercises 23.5.1 and 23.5.2 to help you write the proof. The solution to Exercise 23.5.1 mentions a new substitution lemma. If you add a trivial way to satisfy the new judgment (supporting type variables), then you can use SASyLF's built-in "by substitution" justification. The proof cannot be done by induction over the typing derivation because of the inability to use "exchange" in the T-Abs case. You are encouraged to use "by substitution" also for the normal substitution lemma for practice. The Google project wiki pages have some documentation.

3 Programming

For this section, use the `fullomega` type checker. You should copy the Church encoding of pairs and lists from test.f in the fullomega checker directory and then solve the following:

1. Exercise 23.4.2$\frac{1}{2}$ [$\star$]: Write a recursive `sum` function with type: `sum : (List Nat)` $\to$ `Nat`
2. Write another implementation of `sum` that is *not* recursive but which has the same type and the same behavior. (You are permitted to still use a recursive `plus`.)
3. Exercise 23.4.11$\frac{1}{2}$ [$\star\star$]: Write a *non-recursive* `map` using the Church encoding of lists. It should have the same type as on page 346.
4. Exercise 24.2.5$\frac{1}{2}$ [$\star\star\star$]: The `List` type defined on page 351 exposes the internal representation. For example, we couldn't substitute an implementation using recursive types. The following definition fixes this problem

```
OOList = lambda X.
           {Some R,{state:R, nil:R,
                    isnil: R->Bool,
                    cons: X->R->R,
                    head: R->X,
                    tail: R->R}};
```

   (a) Write a term `oonil` that has type $\forall$`X.OOList X`; use the pre-existing `List X` definition.
   (b) Write definitions of `ooisnil`, `oocons`, `oohead`, `ootail` so that they have the following types:

```
ooisnil : ∀X. (OOList X) → Bool
oocons : ∀X. X → (OOList X) → (OOList X)
oohead : ∀X. (OOList X) → X
ootail : ∀X. (OOList X) → (OOList X)
```

   (c) Write `oomap` to use these primitives (you may assume `fix`). Test your program by running

```
oohead[Bool]
  (oomap[Int][Bool] iseven
    (oocons[Nat] 1 (oocons[Nat] 2 (oonil[Nat]))))
```

Leave all your code in list.f.