Types & Programming Languages
Exercises 6

These exercises are based on the material in Lectures 17, 18 and 19.

1. For each of the following terms, use the constraint typing rules and then unification to work out its principal type.

   (a) $\lambda x : X.((\lambda y : Y.y + 1)x)$

   (b) $\lambda x : X.\lambda y : Y.\mathsf{if}\ x(y)\ \mathsf{then}\ y\ \mathsf{else}\ (\lambda z : Z.z)$

   (c) $\lambda x : X.\lambda y : Y.\lambda z : Z.x(y(z))$

2. We can extend our simply typed lambda calculus with lists, described informally as follows. For every type T there is a type $\mathsf{List}\ T$ which is the type of lists whose elements are of type T. The empty list is represented by nil and there is an operator cons for list construction: $\mathsf{cons}(h, t)$ is the list whose head is h and whose tail is t (so t is a list and h is an expression of suitable type to be part of this list). There needs to be a case construct for lists, which analyses a list and specifies code for the empty (nil) and non-empty (cons) possibilities. All of this is supposed to be similar to lists in Haskell.

   (a) Specify the syntactic extensions, including the syntax of case expressions.

   (b) Define call-by-value reduction rules for list expressions. Remember to specify what are the values.

   (c) Define typing rules for nil, cons and case.

   (d) Define suitable constraint typing rules for nil, cons and case.

   (e) Extend the unification algorithm so that it can handle types involving the List constructor.

   (f) Give an example of a function on lists, and work out its principal type scheme.