Problem 1 - Typing

In lecture 10, we introduced type inference rules for the polymorphic cons function and the polymorphic empty list nil, together with their representation as type expressions. Note that our lists must have elements of the same type, which is not what the corresponding Scheme functions require.

1. Give the corresponding type inferences rules for polymorphic car and cdr, and their type expressions.

2. Use the inference rules from the lecture and your inference rules to construct the type of

(a) (car (cdr (cons 1 (cons 2 nil))))
(b) (lambda (x) (cdr (cdr (car x))))
(c) (lambda (x) (cons (cdr x) nil))

Problem 2 - Unification

Lecture 10, page 10 introduces a simple type expression language consisting of type expression with basic types (char, integer, typeError) and composite types (pointer, array). Assume that your type language also includes type variables, represented as single literals (e.g.: 'a', 'b', ...).

Sketch a UNIFY function that takes two type expressions as input and produces the most general unifier, or reports that such a unifier does not exist. Recursive data types are not allowed.

Examples:

UNIFY( char, pointer(a) ) = ‘does not unify’
UNIFY( pointer(a), pointer(pointer(b))) = { a->pointer(b) }
UNIFY( array(1...a, pointer(char)), array(1...100, pointer(b))) =
   { a->100, b->char }
UNIFY( pointer(a), a ) = ‘does not unify’
Problem 3 - Static vs. Dynamic Typing

Assume that $A$ is declared as follows:

$A$: array(1:100) of integer

1. State a condition under which a compiler at compile time is able to prove that the array reference $A(k)$ will be in bound, i.e., that no out-of-bound access is possible (static typing).

2. For the general case, give the ILOC code that a compiler needs to generate in order to ensure that reference $A(k)$ will not result in an out-of-bound access (dynamic typing).