# Homework Assignment #4

CSE505

Due Monday, November 4 at 2pm

In this assignment, you are going to add *maps* to the simply-typed lambda calculus. The new cases of the language's syntax are as follows:

$$
\begin{array}{lll}
T & ::= & \ldots \mid \texttt{map}\ T_{key}\ T_{val} \\
e & ::= & \ldots \\
 & \mid & \texttt{map}\ \{e_{key_i} \Rightarrow e_{val_i}\ ^{i \in 1..n}\} \\
 & \mid & \texttt{update}\ e_{key} \Rightarrow e_{val}\ \texttt{in}\ e_{map} \\
 & \mid & \texttt{lookup}\ e_{key}\ \texttt{in}\ e_{map}\ \texttt{else}\ e_{absent}
\end{array}
$$

A map is an association list, i.e., a finite function mapping a *key* to its associated *value*. The type $\texttt{map}\ T_{key}\ T_{val}$ is the type of maps whose keys have type $T_{key}$ and whose values have type $T_{val}$. The expression $\texttt{map}\ \{e_{key_i} \Rightarrow e_{val_i}\ ^{i \in 1..n}\}$ creates a new map initialized with the given key-value pairs.[1] This expression is evaluated by evaluating each of its sub-expressions from left-to-right and then producing the associated *map value*. The expression $\texttt{update}\ e_{key} \Rightarrow e_{val}\ \texttt{in}\ e_{map}$ updates a map with a new key-value pair. This expression is evaluated by evaluating each of its sub-expressions from left-to-right and then producing a new map that is identical to the old one except that it additionally contains the new key-value pair. Finally, the expression $\texttt{lookup}\ e_{key}\ \texttt{in}\ e_{map}\ \texttt{else}\ e_{absent}$ looks up a key $e_{key}$ in the map $e_{map}$ and returns the associated value. If the key is not in the map, then $e_{absent}$ is evaluated and its value is returned instead.

You may assume that key values can be compared for equality using regular equality of values in the semantics, e.g., using the same metavariable in an inference rule, or comparing two metavariables using "=". You may handle the issue of a map that contains multiple key-value pairs for the same key any way you'd like. Probably the easiest way is to allow duplicate keys in map values and have lookup simply choose among them non-deterministically.

Here's a small example of how maps may be created and manipulated (assuming an ML-like host language):

```
let val m = (update "x" => 3 in (update "y" => 4 in (map {})))
in (lookup "y" in m else 0) + (lookup "z" in m else 2) end
```

In this example, the variable `m` has type `map string int`, and the entire expression evaluates to `6`.

Assignment:

a. Provide the new cases of the grammar for the values of the simply-typed lambda calculus.

b. Provide the new rules of the dynamic semantics (in the form of a small-step operational semantics), which define how the three new kinds of expressions are evaluated.

c. Provide the new typing rules, which define how the three new kinds of expressions are typechecked.

[1] The value of *metavariable* $n$ may be 0, in which case no pairs are provided.

d. (a) State (no proof necessary) any new parts of the Canonical Forms lemma that are necessary in the proof cases below.

(b) Add the case relevant to the "update" expression to the proof of the Progress theorem:

**Theorem** (Progress): If $\vdash e : T$, then either $e$ is a value or there exists $e'$ such that $e \longrightarrow e'$.

(c) Add the case relevant to the "update" expression to the proof of the Type Preservation theorem:

**Theorem** (Type Preservation): If $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T$.