SC-T-501-FMAL Programming languages, Assignment 3
Spring 2020
Due 17 March 2020 at 23:59

1. Write F# functions `fun1` and `fun2` with the following types:

   ```
   fun1 : ('a -> 'b) -> 'a -> 'a * 'b
   fun2 : (('a -> 'b) -> ('a -> 'b)) -> 'a -> 'b
   ```

   Do not use exceptions, and do not give any explicit type annotations.

2. For each of the following pairs of types, say whether they can be unified or not. If they can be unified, list the assignments of type variables that need to be made to achieve this.

   - `'a -> 'b` and `int -> int`
   - `'a -> 'b -> 'c` and `'d -> int`
   - `'a -> 'a` and `('b -> 'c) -> (int -> 'c)`
   - `'a -> int` and `('a -> 'a) -> int`

   In the last pair of types, `'a` is the same variable in both types. (You are not allowed to rename apart the occurrences of `'a` in the first resp. second type.)

   (Provide the answers as comments in the appropriate place in the file.)

3. What are the normal forms and types of the following terms in the simply typed (i.e., non-polymorphically typed) lambda-calculus? Assume that f is assigned the type $X \to X$ where X is a type variable. (It is not universally quantified, you cannot specialize X.)

   - $(\lambda x.\, x)\, f$
   - $(\lambda g.\, \lambda x.\, f\,(g\,x))\,(\lambda y.\, f\,y)$
   - $\lambda h.\, (\lambda g.\, g\,(g\,f))\,h$
   - $((\lambda h.\, \lambda k.\, k\,h)\,(\lambda y.\, f\,(f y)))\,(\lambda g.\, \lambda x.\, g\,(g\,x))$
   - $((\lambda h.\, \lambda f.\, f\,h)\,(\lambda x.\, f\,(f x)))\,(\lambda g.\, \lambda x.\, g\,(g\,x))$

   Be careful to avoid capture when normalizing the terms by renaming the bound variables where necessary.

   (Provide the answers as comments in the appropriate place in the file.)

4. The evaluator in the provided F# file copies the entire environment into every closure, including the variables that are not used inside the function. Change the implementation of `eval` so that closures only include the variables that actually occur in the function.

5. The type `expr` in the provided F# file contains a constructor `Annot`, which allows expressions to contain type annotations. That is, among all our expression forms, we have the expression form `e : t` where `t` is a simple type (it cannot contain forall). The idea is to say that the expression `e` must admit the type `t`.

   The provided implementation of type inference just ignores the type annotations.

   Change the definition of the function `infer` so that these annotations are respected. It should raise an exception if the expression cannot be assigned the given type, and should specialize type variables appropriately if it can be given this type.