T-501-FMAL Programming languages, Assignment 3
Spring 2021
Due 19 March 2021 at 23:59

1. What would the result of evaluating the following expression be under (i) the static scope rule; (ii) the dynamic scope rule?

   ```
   let y = 3 in
     let f g = g 2 + g y in
       let y = f (fun x -> x + 4) in
         f (fun x -> x + y)
   ```

2. Write F# functions `option_fun` and `list_fun` with the following types:

   ```
   list_fun   : ('a -> 'a) -> 'a -> 'a list   -> 'a
   option_fun : ('a -> 'a) -> 'a -> 'a option -> 'a
   ```

   F# should report these types, not more general ones like `'a -> 'b -> c -> d` or more specific ones like `(int -> int) -> int -> int list -> int`. Do not use type annotations or exceptions. Your functions must terminate on all inputs.

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

   You are not allowed to rename apart the occurrences of type variables. Be careful not to create infinite types (for example, `'a` and `'a list` cannot be unified).

   E.g., the pairs `'a list * 'b` and `'c * (int -> 'c)` unify, and this achieved by taking `'c ↦ 'a list` and `'b ↦ int -> 'a list`. The pairs `'a -> int` and `bool -> 'a` do not unify.

   (i) `'a -> 'a` and `'a -> int list`

   (ii) `'a -> 'b` and `'a -> int list`

   (iii) `(int -> int) -> (int -> int)` and `'a -> 'a`

   (iv) `'a list -> 'a list` and `'b -> 'b`

   (v) `'a list -> 'a` and `'b -> 'b`

   (vi) `('a -> 'b) -> 'c` and `'d -> 'e list`

The file `Assignment3.fs` contains a partial implementation of a language with `floats`, vectors (lists) of `floats`, and (first-order) functions. The syntax includes:

- constant floats (`NumF`) and vectors (`Vect`);
- addition (`Plus`) of either two `floats`, yielding a `float`; or of two vectors of the same length, yielding a vector of that length;
- taking the average (`Average`) of the elements of a vector, yielding a `float`;
- multiplication (`Scale`) of a `float` and a vector, yielding a vector of the same length;
- if expressions (`IfPositive`) of the form `if e > 0 then e1 else e2`, where both branches are required to have the same type, and `e` is a `float`;
- variables (`Var`), function calls (`Call`) and non-recursive function declarations (`LetFun`).

(Ignore `LetFunNoGeneralize` for the first few problems.) Look at the definition of `eval` for the exact behaviour of the expressions.

Functions can only accept vectors as arguments, but they can be polymorphic in the length of the vector. For example, `LetFun ("f", "x", Var "x", Var "f")` is the expression `let f x = x in f`, which has type `Vector ('m) -> Vector ('m)` where `'m` is a variable for the length of the vector. Vectors can only be added together when they have the same length, so

```
LetFun ("p", "x", Plus (Var "x", Vect [1.2; 3.4; 5.6]), Var "p")
```

has type `Vector(3) -> Vector(3)`,

```
LetFun ("f", "x", LetFun ("g", "y", Plus (Var "x", Var "y"), Var "g"), Var "f")
```

has type `Vector ('m) -> (Vector ('m) -> Vector ('m))`, and `Plus (Vect [1.2], Vect [3.4; 5.6])` is ill-typed.

The types of this language are:

- `Float`: floating-point numbers;
- `Vector (l)`: vectors of length `l`;
- `Fun (l, t)`: the function type `Vector (l) -> t`.

where `l` is a length (either an integer or a length variable `'m`, `'n`, . . .).

4. Implement a unification function `unify : typ -> typ -> unit`. This should ensure that lengths of vectors can be unified (you can use `unifyLength` to do this).
5. Complete the definition of type inference, by implementing the cases for `Plus`, `Average` and `Scale` in the function `infer`. In addition to `unify`, some functions in `Assignment3.fs` will be useful: `ensureFloat`, `ensureVector` and `ensureFloatOrVector` ensure that a type is `Float`, `Vector`, or either of the two.
6. `LetFunNoGeneralize` is the same as `LetFun`, except that type inference does not generalize the length variables. Use `LetFunNoGeneralize` to define an expression `no_generalize` such that type inference fails on `no_generalize`, but would have succeeded (in the empty type environment) if you had used `LetFun` instead.