ASSIGNMENT 7 — TYPES IN THEORY AND PRACTICE

COMP 3010 — ORGANIZATION OF PROGRAMMING LANGUAGES

1. Type Systems

Recall the grammar of the conditional arithmetic language we have been studying in this class. The two different syntactic types (A and B) in the language's grammar can be merged into this single type of general expression (M):

$$n ::= 0 \mid 1 \mid 2 \mid 3 \mid \dots$$
$$b ::= \mathit{true} \mid \mathit{false}$$
$$M ::= \underline{n} \mid \underline{b} \mid \texttt{plus}(M, M) \mid \texttt{minus}(M, M) \mid \texttt{geq?}(M, M) \mid \texttt{if}(M, M, M) \mid \dots$$

After the merge, new expressions become syntactically valid, like $\texttt{plus}(\mathit{true}, \mathit{false})$, which don't have a meaningful interpretation as a program because evaluation doesn't know how to produce an answer! regain our sanity and rule out these sorts of errors, we can use types T to distinguish what results an expression is expected to return, which have only two cases for this simple language:

$$T ::= \texttt{num} \mid \texttt{bool}$$

Here are some of the relevant typing rules that confirm that a particular expression M belongs to the type T, written as $M : T$.

$$\frac{}{\underline{n} : \texttt{num}} \qquad \frac{}{\underline{b} : \texttt{bool}}$$

$$\frac{M_1 : \texttt{num} \quad M_2 : \texttt{num}}{\texttt{plus}(M_1, M_2) : \texttt{num}} \qquad \frac{M_1 : \texttt{num} \quad M_2 : \texttt{num}}{\texttt{minus}(M_1, M_2) : \texttt{num}} \qquad \frac{M_1 : \texttt{num} \quad M_2 : \texttt{num}}{\texttt{geq?}(M_1, M_2) : \texttt{bool}}$$

$$\frac{M_1 : \texttt{bool} \quad M_2 : T \quad M_3 : T}{\texttt{if}(M_1, M_2, M_3) : T}$$

Exercise 1. What is the type of $\texttt{if}\,(\texttt{geq?}(\underline{3}, \underline{5}), \texttt{geq?}(\texttt{minus}(\underline{3}, \underline{5}), \underline{6}), \texttt{geq?}(\texttt{plus}(\underline{3}, \underline{5}), \underline{6}))$? Show that your choice of type T is correct by drawing a type checking tree following the above rules exactly, so that your tree has the conclusion

$$\texttt{if}\,(\texttt{geq?}(\underline{3}, \underline{5}), \texttt{geq?}(\texttt{minus}(\underline{3}, \underline{5}), \underline{6}), \texttt{geq?}(\texttt{plus}(\underline{3}, \underline{5}), \underline{6})) : T$$

at the bottom of the tree (plugging in your choice of `num` or `bool` in for T).

Hint: For example, the type-checking tree showing $\texttt{geq?}(\underline{1}, \texttt{minus}(\underline{2}, \underline{1})) : \texttt{bool}$ looks like this (notice how the goal $\texttt{geq?}(\underline{1}, \texttt{minus}(\underline{2}, \underline{1})) : \texttt{bool}$ appears exactly as-is on the bottom of the tree):

$$\frac{\frac{}{\underline{1} : \texttt{num}} \quad \frac{\frac{}{\underline{2} : \texttt{num}} \quad \frac{}{\underline{1} : \texttt{num}}}{\texttt{minus}(\underline{2}, \underline{1}) : \texttt{num}}}{\texttt{geq?}(\underline{1}, \texttt{minus}(\underline{2}, \underline{1})) : \texttt{bool}}$$

Date: Spring 2022.

The grammar of the λ-calculus already combines several different types of expressions in the same syntactic group M:

$$M ::= x \mid M\ M \mid \lambda x.M$$

The types (T) of λ-expression could be primitive types X or a function $T_1 \to T_2$ which takes T_1's as arguments and returns T_2's as results.

The first challenge in checking the type of λ-expressions is knowing what type of values to expect for the free variables in it. This challenge is solved by using a typing environment $\Gamma = x_1 : T_1, x_2 : T_2, x_3 : T_3, \dots, x_n : T_n$ that assigns a specific type to all free variables that might appear in an expression. Here is a definition of the system of checking simple types of λ-calculus expressions (where the empty environment ϵ is often not written, so that $x : T, y : T'$ means $(\epsilon, x : T), y : T'$)

$$T ::= X \mid T \to T \qquad\qquad \Gamma ::= \epsilon \mid \Gamma, x : T$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash M : T' \to T \quad \Gamma \vdash M' : T'}{\Gamma \vdash M\ M' : T} \qquad \frac{\Gamma, x : T' \vdash M : T}{\Gamma \vdash \lambda x.M : T' \to T}$$

Exercise 2. Use the typing rules for the λ-calculus to check that $(\lambda x.\lambda y.(x\ (z\ y)))$ has the type $(\texttt{string} \to \texttt{num}) \to (\texttt{num} \to \texttt{num})$ whenever the free variable z has the type $\texttt{num} \to \texttt{string}$. In other words, draw a type checking tree following the rules above with $z : \texttt{num} \to \texttt{string} \vdash (\lambda x.\lambda y.(x\ (z\ y))) : (\texttt{string} \to \texttt{num}) \to (\texttt{num} \to \texttt{num})$ as the *exact* conclusion at the bottom of the tree.

Hint: The typing tree that checks the application $y\ z$ has the type T_2 in an environment where y has type $T_1 \to T_2$ and z has type T_1 looks like this:

$$\frac{\dfrac{(y : T_1 \to T_2) \in y : T_1 \to T_2, z : T_1}{y : T_1 \to T_2, z : T_1 \vdash y : T_1 \to T_2} \quad \dfrac{(z : T_1) \in y : T_1 \to T_2, z : T_1}{y : T_1 \to T_2, z : T_1 \vdash z : T_1}}{y : T_1 \to T_2, z : T_1 \vdash y\ z : T_2}$$

Exercise 3. For both of the following scenarios, say which form of polymorphism is solved by generics (a.k.a. parametric polymorphism) and which one is solved by subtyping. In both cases, explain why that form of polymorphism helps solve the problem.

(1) You want to write a function that takes two objects and invokes the `age` method on each of them to decide if the first one is younger (has a smaller `age`) than the second. You know that the `Person` class of objects all have a an implementation of the `age` method which returns a number, but your function should work on other classes of objects that also have an `age` method.
(2) You want to write a function that takes a list/array of elements and reverses it. You don't need to inspect operate on any of the elements directly, but only copy them into a new list/array where they appear in the opposite order.

2. Types in SML

Exercise 4. Do *Concepts In Programming Languages* Exercise 6.1 parts on understanding SML types (page 156).

Hint 1: In SML, the arithmetic operations `+` and `*` are overloaded and can be applied to different types of numbers (including the whole numbers of type `int` like `3` and fractional floating-point numbers of type `real` like `3.5`) as long as both arguments are *exactly the same type of number*. The result of `x+y` (and similarly `x*y`) will be *the same type of number* as its two arguments `x` and `y`. In contrast, the fractional division operation `/` only applies to fractional floating-point numbers of type `real`, and gives back a number of the same type `real`.

Hint 2: Remember that the syntax of an inline function in SML is `fn x => ...` (analogous to $\lambda x. \dots$ in the λ-calculus). A function type in SML is written using an ASCII arrow `->`, which is a binary operation that associates to the right, so that `'a -> ('b -> 'c)` is the same type as `'a -> 'b -> 'c`.

Exercise 5. Do *Concepts In Programming Languages* Exercise 6.7 on using type inference to help identify bugs (page 159).

Hint: The `append` function is meant to take two lists as arguments and return a single containing *all* of the elements from both of them in the same order they appeared in, beginning with the elements of the first list and then ending with all the elements of the second list. For example, `append([1,2,3,4],[3,4,5,6])` should be `[1,2,3,4,3,4,5,6]`. As such, the type of `append` is supposed to be `'a list * 'a list -> 'a list` (since the elements in the lists could have any type `'a` as long as they are the same).

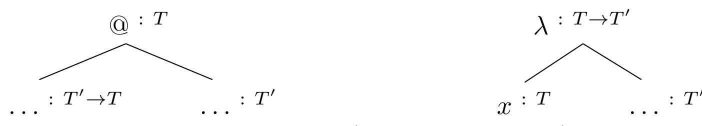
What is the type of the given `append` function defined by the code in *CIPL* Exercise 6.7, why is it not the same as the expected type above, and what does this tell you about what the buggy code does differently compared to what it is supposed to do?

Exercise 6. Do *Concepts In Programming Languages* Exercise 6.5 on calculating the type of a parse graph (pages 157 & 158).

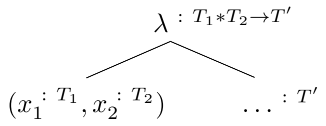
Hint: To calculate the type of the function corresponding to the syntax tree, follow the procedure we used in class:

(1) Annotate *every* node of the syntax tree with a type.
   - Annotate constants (such as the number `2` which always has the type `int` or the operator `+` used at the type `int -> (int -> int)`) with their known type.
   - Annotate the place where function parameters and local variables are introduced with a unique type variable (`'a`, `'b`, `'c`, *etc.*). Then, when those same variable names are referred to later, use the same type variable as when they were introduced.
   - All other syntax nodes (such as applications and λs) get annotated with a unique type variable different from all the other type variables you use in the tree.

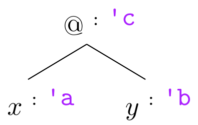
(2) Start identifying the typing constraints created by each node in the syntax tree, and write those constraints forcing the generic type variables introduced in the previous step to be equal to some more specific type.
   - For example, the two main forms of constraints created by function application and abstraction have the shapes (T and T' might be different types, but repetitions of T stand for the *same* type appearing in multiple places):



λs binding a pair of two parameters (as in this exercise) look like this:



- When you find a syntax node currently has a placeholder type variable (such as `'a`) as its type, but one of the above constraints forces the type to be something more specific, then write that constraint as an equation that assigns the more specific type to the placeholder. For example, in



  The application constraint above requires that x have a function type written with an arrow with the type of the argument y on the left and the type of the application @ on the right. In other words, this part of the syntax tree forces the following constraint on the type `'a`:

  `'a` = `'b -> 'c`

- When you discover a new constraint on a placeholder type variable (like the example above on `'a`), you can make your life easier by crossing out that type variable *everywhere* it appears in the syntax tree and replacing it with the new, more specific type.

(3) There might still be some initial placeholder type variables remaining, even after you have completed step (2) for *every* node in the syntax tree, and managed to solve *all* of the constraints forced on type variables. That's okay! Those remaining type variables can be left *generic*, since the code evidently does not care what specific types they stand for. Leave any such remaining type variables in the type signature of the definition (*Hint:* the final type signature of the definition will be the type annotating the top of the syntax tree), giving you a *polymorphic* type for the code.