EXTRA CREDIT 1 — TYPE INFERENCE

COMP 3010 — ORGANIZATION OF PROGRAMMING LANGUAGES

1. Inferring the Type of One Expression

In assignment 8, you implemented a type checker for a language based on the λ-calculus with numbers, booleans, and if-then-else conditionals. Notably, the algorithm you wrote was a type *checker* because it expects to receive the syntax tree of a programs with enough typing annotations added to it so that it will never have to guess.

For example, the type checker might receive the syntax tree representing the function λx : `boolean`. x, where the parameter x has been annotated (say, by the programmer) with the type `boolean`. That means that you can immediately determine that this function has the type `boolean` $\rightarrow$ `boolean`, even though there is nothing in the body of the function which hints that x should be a boolean instead of something else, like a number or a string.

If you were not given any type annotations on function parameters, then what type should you say that the function $\lambda x.x$ has? It receives some value (x) as an argument, and then just returns it as-is without interacting with it. For all practical purposes, x could stand for a boolean, or a number, or a string, and nothing would go "wrong." The way to write down this more generic description specifying the type of this function is $\forall a.\ a \rightarrow a$ which says that the function can accept an argument of any type (a), and will return a result of the *same* type (a) matching the argument it was given.

These extra credit exercises ask you to generalize the algorithm you completed in assignment 8 to be able to *infer* the type of an expression that has *absolutely no* typing annotations in it, and to be able to assign the *most general* type to functions which can operate over many types of parameters. If you choose, you may implement your solutions to these exercises in any of the languages we have explicitly used so far in class (that is, Ruby, Scheme, or Standard ML). Get permission from the instructor first if you wish to implement your solutions in another language.

Extra Credit Exercise 1. Implement the type inference algorithm discussed in *Concepts in Programming Languages* chapter 6.3, also known as Hindley-Milner's

Date: Spring 2022.

algorithm J, for the same language as in assignment 8, *except* that function parameters are written *without* any type annotations:

$$
\begin{aligned}
n &::= 0 \mid 1 \mid 2 \mid 3 \mid \ldots \\
b &::= \mathit{true} \mid \mathit{false} \\
x &::= \text{any symbolic identifier name} \\
M &::= n \mid b \mid x \mid M\ M \mid \lambda x.M \mid \mathit{if}\ M\ \mathit{then}\ M\ \mathit{else}\ M
\end{aligned}
$$

$$
\begin{aligned}
a &::= \text{any symbolic identifier name} \\
T &::= a \mid \mathit{number} \mid \mathit{boolean} \mid T \to T
\end{aligned}
$$

where a stands for an *unknown type variable*. Notice how the x in $\lambda x.M$ has no type annotation, contrary to assignment 8.

Your type inference algorithm should follow these (partially informal) typing rules for this part

$$\Gamma ::= x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n$$

$$\frac{}{\Gamma \vdash n : \mathit{number}} \qquad \frac{}{\Gamma \vdash b : \mathit{boolean}}$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash M_1 : T_1 \quad \Gamma \vdash M_2 : T_2 \quad a = \texttt{fresh\_type\_variable}() \quad \texttt{unify}(T_1, T_2 \to a)}{\Gamma \vdash M_1\ M_2 : a}$$

$$\frac{a = \texttt{fresh\_type\_variable}() \quad \Gamma, x : a \vdash M : T}{\Gamma \vdash \lambda x.M : a \to T}$$

$$\frac{\Gamma \vdash M_1 : T_1 \quad \texttt{unify}(T_1, \mathit{boolean}) \quad \Gamma \vdash M_2 : T_2 \quad \Gamma \vdash M_3 : T_3 \quad \texttt{unify}(T_2, T_3)}{\Gamma \vdash \mathit{if}\ M_1\ \mathit{then}\ M_2\ \mathit{else}\ M_3 : T}$$

For more details, see `https://en.wikipedia.org/wiki/Hindley\OT1\textendashMilner_type_system`.

In particular, your approach to implementing this algorithm will need to be able to accomplish these tasks:

(1) Be able to generate *fresh new type variables* which have never appeared before during the same run of the type inference algorithm. This is signaled by the use of the `fresh_type_variable`() operation used in two of the inference rules above. In your code, every call to `fresh_type_variable`() should return a unique type variable with a new name that it has never returned before.

(2) Be able to *unify* two types, signaled by the use of the `unify` operation used in two of the inference rules above. Intuitively, $\texttt{unify}(T_1, T_2)$ is an operation that may either succeed or fail. If a typing rule requires $\texttt{unify}(T_1, T_2)$ as a premise, and $\texttt{unify}(T_1, T_2)$ fails, then that should raise a type error. Otherwise, if $\texttt{unify}(T_1, T_2)$ succeeds, then you should continue type inference and *remember* any constraints that might be imposed on type variables caused by unification. For example:

- `unify`(a, b) with two unknown type variables a and b always succeeds, and imposes the constraint that the two variables are equal ($a = b$) for the remainder of type inference.
- `unify`(a, T) and `unify`(T, a) for any other type T also succeed, and imposes the constraint that $a = T$.
- `unify`$(\mathit{number}, \mathit{number})$ and `unify`$(\mathit{boolean}, \mathit{boolean})$ both succeed, and impose no additional constraints that need to be remembered. In contrast, `unify`$(\mathit{number}, \texttt{boolean})$ and symmetrically `unify`$(\mathit{boolean}, \mathit{number})$ *always fails*, and raises a type error.
- `unify`$(T_1 \to T_2, T_1' \to T_2')$ succeeds *only if both* `unify`(T_1, T_1') and `unify`(T_2, T_2') succeed, and you need to remember all the constraints imposed by both `unify`(T_1, T_1') and `unify`(T_2, T_2') for inferring the type of the remainder of the expression.
- The function type $T_1 \to T_2$ *always fails* to unify with the primitive types *number* and *boolean*.

Note that the order of arguments to `unify` doesn't matter: the result of `unify`(T_1, T_2) should be the same as `unify`(T_2, T_1), and impose the same constraints for what type each variable should be equal to.

For help remembering the constraints setting variables equal to more specific types, consider using the union-find data structure: `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`.

(3) After you have finished inferring the type of the entire expression without raising any Type Errors, you will need to generate the final type by applying the constraints gathered from unification to plug in more specific types assigned to initially-unknown type variables. For example, if you started with the type $a \to b$ and found the constraints

$$a = \mathit{number} \qquad\qquad b = \mathit{number} \to \mathit{boolean}$$

then you should return the final type $\mathit{number} \to (\mathit{number} \to \mathit{boolean})$. If instead you started with the type $a \to b$ and found the constraints

$$a = b \qquad\qquad b = \mathit{number} \to \mathit{boolean}$$

then you should return the final type $(\mathit{number} \to \mathit{boolean}) \to (\mathit{number} \to \mathit{boolean})$.

Any unknown type variables remaining after you plugged in every constraint should be explicitly listed with a $\forall$ that names the generic variables that will appear in the final return type. So if you started with the type $a \to b$ and found only the constraint

$$b = a \to a$$

then you should return the final type $\forall a.\ a \to (a \to a)$. If instead you started with the type $a \to b$ and found the constraint

$$a = c \to d \qquad\qquad b = c \to e$$

then you should return the final type $\forall c.\ \forall d.\ \forall e.\ (c \to d) \to (c \to e)$.

2. Referring to Polymorphic Variables

The type system in section 1 does not let you actually take advantage of polymorphism in named variables that come in from the outside. For example, you may begin to infer the type of an expression that refers to a free variable x (defined

previously and imported from outside of the expression) by assigning a type like $number \to number$ to x in the initial environment, which means that x can only be used as a function converting a number into another number. Even if you assign a type like $a \to a$ to x in the initial environment, the rule which looks up x's type still forces you to assume that x has *exactly* the type $a \to a$ for some fixed a.

Extra Credit Exercise 2. Generalize your type inference algorithm to allow for free variables in the environment Γ to be assigned *polytypes* (short for *polymorphic types*): that is, types with zero or more generic type variables (introduced by a $\forall a$ meaning "for all choices of type a"). Environments where variables are assigned polytypes (P) fit into this grammar:

$$\begin{aligned} T &::= a \mid number \mid boolean \mid T \to T \\ P &::= \forall a.P \mid T \\ \Gamma &::= x_1 : P_1, x_2 : P_2, \ldots, x_n : P_n \end{aligned}$$

When looking up a variable that has been assigned a polytype, you must *instantiate* the generic type variables (introduced by $\forall$s) with some chosen types T. For example, if x is assumed to have the polytype $\forall a.\ a \to a$ in the environment, then you can plug in any type you want, like *number*, for the a introduced by the $\forall$, to say that x also has the specialized type $number \to number$. Later on in the same expression, you might use x again and instead choose to instantiate a with *boolean*, to say that x also has the specialized type $boolean \to boolean$.

In the most general case, you can always instantiate a polytype like $\forall a.\ a \to a$ by plugging in a currently unknown type b in for a to get the type $b \to b$, and then figure out what b should have been later based on how it is used. The typing rule that matches this general case for instantiating a variable with a polymorphic type is:

$$\frac{(x : \forall a_1.\ \forall a_2.\ \ldots \forall a_n.T) \in \Gamma \quad \text{for } 1 \le i \le n, \quad b_i = \texttt{fresh\_type\_variable}()}{\Gamma \vdash x : [b_1/a_1][b_2/a_2]\ldots[b_n/a_n]T}$$

where you have plugged in fresh new type variables $b_1, b_2, \ldots b_n$ for the generic type variables $a_1, a_2, \ldots a_n$ used in the polytype $\forall a_1.\ \forall a_2.\ \ldots \forall a_n.T$ assigned to x in Γ. Note that renaming one type variable a with another b inside of T is written $[b/a]T$, and can be implemented as a simple search-and-replace over a monomorphic type T following these rules:

$$\begin{aligned} [b/a]a &= b \\ [b/a]a' &= a' && \text{(if } a \neq a'\text{)} \\ [b/a]number &= number \\ [b/a]boolean &= boolean \\ [b/a](T \to T') &= ([b/a]T) \to ([b/a]T') \end{aligned}$$

Generalize the type inference algorithm you wrote in extra credit exercise 1 to use this typing rule for instantiating variables of polymorphic types, instead of the previous rule in extra credit exercise 1 which assumes x has a monomorphic type.

3. Binding Polymorphic Variables

extra credit exercise 1 is about inferring the (polymorphic) type of one expression, and extra credit exercise 2 is about referring to variables standing in for polymorphic

values imported from the outside. This section connects these two together, and extends the language you've implemented with the ability to create and define *new* polymorphic values *within* the language.

Extra Credit Exercise 3. Add an explicit let-expression construct to your language, which binds a variable name to a value which might have a *polymorphic type*. This expression can be written nicely in abstract syntax as this new form in the grammar of expressions:

$$M ::= \cdots \mid let\ x = M\ in\ M$$

And the rule for typing this let-expression is

$$\frac{\Gamma \vdash M_1 : T_1 \quad \{a_1, a_2, \ldots, a_n\} = FV(T_1) - FV(\Gamma) \quad \Gamma, x : \forall a_1.\ \forall a_2.\ \ldots \forall a_n.\ T_1 \vdash M_2 : T_2}{\Gamma \vdash let\ x = M_1\ in\ M_2 : T_2}$$

where $FV(T_1)$ returns the set of *free type variables* found in a monomorphic type, defined as:

$$\begin{aligned} FV(a) &= \{a\} \\ FV(number) &= \{\} \\ FV(boolean) &= \{\} \\ FV(T_1 \to T_2) &= FV(T_1) \cup FV(T_2) \end{aligned}$$

To implement this rule, whenever you need to infer the type of the expression $let\ x = M_1\ in\ M_2$ do the following:

(1) Infer the type of the expression M_1 using your type inference algorithm from extra credit exercises 1 and 2. You may want to introduce an new set of constraints introduced from inferring M_1's type in order to keep them separate from the any previous constraints in the next step.
(2) After you have finished inferring the type of M_1, plug in all solutions to the constraints introduced and during this phase of type inference, and abstract over (with ∀s) all remaining type variables that were generated during the inference of M_1's type. *CAUTION:* make sure not to abstract over any type variables that were generated previously in some other phase of type inference; those variables may be referred to in the outer Γ and *cannot* be generic in the type of M_1.
(3) Once you have determined the polymorphic type $\forall a_1\ \forall a_2.\ \ldots \forall a_n.\ T_1$ for M_1, add the type assignment $x : \forall a_1\ \forall a_2.\ \ldots \forall a_n.\ T_1$ to the environment used to infer the type of M_2.

Extra Credit Exercise 4. Regular let-expressions let you introduce names for polymorphic expressions which refer *only* to previously-defined values. More interesting are *recursive* bindings which give a name to code that can refer to previous definitions *as well as itself*. During type inference, recursive code might need to refer to itself while you check its type, but you don't know what the final type will be until it's done. To cut the Gordian knot, you can just start with a placeholder type that will be filled in as you infer the type of the recursive definition.

Extend the grammar of your language with recursive expression

$$M ::= \cdots \mid rec\ x = M\ in\ M$$

and extend your type inference algorithm to figure out the type of recursive expressions following this typing rule:

$$\frac{\begin{array}{l} a = \texttt{fresh\_type\_variable}() \\ \Gamma, x : a \vdash M_1 : T_1 \\ \texttt{unify}(a, T_1) \\ \{b_1, b_2, \ldots, b_n\} = FV(T_1) - FV(\Gamma) \\ \Gamma, x : \forall b_1.\ \forall b_2.\ \ldots \forall b_n.\ T_1 \vdash M_2 : T_2 \end{array}}{\Gamma \vdash rec\ x = M_1\ in\ M_2 : T_2}$$