COMP3161/9164 23T3 Assignment 2
Type Inference for Polymorphic `MinHS`

Version 1.3.2

Marks : 17.5% of the overall mark

Due date: **~~Sunday 19th~~ Tuesday 21st November 2023, 11:59 PM Sydney time**

Overview

In this assignment you will implement type inference for `MinHS`. The language used in this assignment differs from the language of Assignment 1 in two respects: it has a polymorphic type system, and it has aggregate data structures.

The assignment requires you to:

- (100% marks) Implement the type inference algorithm discussed in the lectures for polymorphic `MinHS` with sum and product data types;
- (5% bonus) adjust the type inference pass to include various simple syntax extensions from Assignment 1;
- (5% bonus) adjust the type inference pass to allow optional type annotations provided by the user;

Each of these parts is explained in detail below.

The `MinHS` parser and evaluator are provided for you. You do not have to change anything in any module other than `TyInfer.hs` (even for the bonus parts).

Your type inference pass should return the inferred type scheme of the top-level binding, `main`.

Your assignment will only be tested on correct programs, and will be judged correct if it produces a correct type for `main` up to α-renaming of type variables, and reordering of quantifiers.

Please use the Ed forum for questions about this assignment.

Submission

Submit your (modified) `TyInfer.hs` using the CSE give system, by typing the command

```
give cs3161 TyInfer TyInfer.hs
```

or by using the CSE give web interface.

1 Task 1

Task 1 is worth 100% of the marks of this assignment. You are to implement type inference for `MinHS` with aggregate data structures. The following cases must be handled:

- the `MinHS` language of the first task of assignment 1 (without n-ary functions, or lists);
- product types: the 0-tuple (aka the `Unit` type) and 2-tuples;

- sum types;
- polymorphic functions

These cases are explained in detail below. The abstract syntax defining these syntactic entities is in `Syntax.hs`. You should not need to modify the abstract syntax definition in any way.

Your implementation is to follow the definition of inference rules provided with this assignment. In particular, you must implement a *variables-in-contexts* version of Hindley-Milner type inference by tracking *definitions* (in addition to declarations) for type variables in the context.

Additional material can be found in the lecture notes on polymorphism, and the reference materials. The *variables-in-contexts* approach is outlined in this specification along with inference rules.

2 Bonus Tasks

These tasks are all optional, and are worth a total of an additional 10%. Marks above 100% are converted to exam marks, at an exchange rate of 1 to 0.15—for example, a mark of 105% yields 0.75 bonus marks for the exam.

2.1 Bonus Task 1: Simple Syntax Extensions

This bonus task is worth an additional 5%. In this task, you should implement type inference for multiple bindings in the one `let` expression, with the same semantics as the extension task for Assignment 1.

You will need to develop the requisite extensions to the type inference algorithm yourself, but the extensions are very similar to the existing rules.

2.2 Bonus Task 2: User-provided type signatures

This bonus task is worth an additional 5%. In this task you are to extend the type inference pass to accept programs containing *some* type information. You need to combine this with the results of your type inference pass to produce the final type for each declaration. That is, you need to be able to infer correct types for programs like:

```
main = let f :: (Int -> Int)
             = recfun g x = x;
       in f 2;
```

You must ensure that the type signatures provided are not overly general. For example, the following program should be a type error, as the user-provided signature is too general:

```
main :: (forall 'a. 'a) = 3;
```

You may assume for simplicity that the user-provided types have distinct type variable names for all bound / free type variables. Your solution should, for example, support programs such as:

```
main :: forall 'a. 'a -> 'a + Int = recfun m y =
    let g :: forall 'b. 'b -> 'a + 'b = recfun f x = Inl y;
    in g 1
```

where the type variable `'a` is in scope for the expression bound to `main` and appears in the user-provided type annotating `g`. All occurrences of `'a` within the bound expression reference the same type variable.

3 Algebraic Data Types

This section covers the extensions to the language of the first assignment. In all other respects (except lists) the language remains the same, so you can consult the reference material from the first assignment for further details on the language.

3.1 Product Types

We only have 2-tuples in `MinHS`, and the `Unit` type, which could be viewed as a 0-tuple.

$$
\begin{array}{lrcl}
\text{Types} & \tau & ::= & \tau \star \tau \\
 & & | & \texttt{1} \\
 & & & \\
\text{Expressions} & e & ::= & (e,\ e) \\
 & & | & \texttt{fst}\ e \mid \texttt{snd}\ e \\
 & & | & \texttt{()}
\end{array}
$$

3.2 Sum Types

Sum types in `MinHS` follow the presentation in the lectures.

$$
\begin{array}{lrcl}
\text{Types} & \tau & ::= & \cdots \mid \tau + \tau \\
 & & & \\
\text{Expressions} & e & ::= & \cdots \\
 & & | & \texttt{Inl}\ e \\
 & & | & \texttt{Inr}\ e \\
 & & | & \texttt{case}\ e\ \texttt{of} \\
 & & & \qquad \texttt{Inl}\ x\ \texttt{->}\ e\ \texttt{;} \\
 & & & \qquad \texttt{Inr}\ y\ \texttt{->}\ e\ \texttt{;}
\end{array}
$$

3.3 Polymorphism

The extensions to allow polymorphism are relatively minor. Three new type forms have been introduced: the `FlexVar t` form, the `RigidVar t` form, and the `Forall t e` form. `FlexVar t` represents a unification variable introduced during type inference. `RigidVar t` represents a fixed type variable introduced by a forall-quantifier. Consult Section 4 for more details on the notational conventions used in this specification and how they relate to the Haskell code. We distinguish between type *schemes* and other types:[1]

$$
\begin{array}{lrcl}
\text{Scheme} & \sigma & ::= & \texttt{forall}\ \overline{\alpha}.\tau \\
\text{Types} & \tau & ::= & \alpha^{\mathrm{F}} \mid \alpha^{\mathrm{R}} \mid \tau \star \tau \mid \tau\ +\ \tau \mid \cdots
\end{array}
$$

Type inference should return a correctly typed top-level binding for `main`. For example, consider the following code fragment before and after type inference:

```
main =
  let f = recfun g x = x;
  in if f True
       then f (Inl 1)
       else f (Inr ());

main :: (Int + 1) =
  let f = recfun g x = x;
  in if f True
       then f (Inl 1)
       else f (Inr ());
```

4 Notational Conventions

In this document, we will use a number of conventions and conveniences to streamline the presentation detailed in Table 1.

[1] Here, and elsewhere in the spec we use $\overline{overlines}$ to mean "sequence of". So, e.g., $\overline{\sigma}$ is a sequence of type schemes.

Concept	Specification Notation	Haskell Notation
Flexible Type Variable	$\alpha^{\mathrm{F}}, \beta^{\mathrm{F}}, \rho^{\mathrm{F}}, \ldots$	`FlexVar` α
Rigid Type Variable	$\alpha^{\mathrm{R}}, \beta^{\mathrm{R}}, \rho^{\mathrm{R}}, \ldots$	`RigidVar` α
Flexible Type Variable Declaration	α	α `:=HOLE`
Flexible Type Variable Definition	$\alpha := \tau$	α `:=Defn` τ
Rigid Type Variable Substitution	$\tau[\alpha^{\mathrm{R}} := \tau']$	`substRigid` $(\alpha$ `=:` $\tau')$ τ
Flexible Type Variable Substitution	$\tau[\alpha^{\mathrm{F}} := \tau']$	`substFlex` $(\alpha$ `=:` $\tau')$ τ
Term Variable Entry	$x : \sigma$	`TermVar` x σ

Table 1: Notations and Conventions in this Specification vs. Haskell code for Assignment

Type variable *names* are ranged over by lowercase greek letters. We distinguish between flexible and rigid type variables by superscripting such names with `F` and `R`, respectively.

Declarations and definitions for flexible type variables appear in typing contexts, see Section 5.1 for the full grammar of contexts.

Substitution for type variables occurring in types is explained in Section 6. Since there are two kinds of type variables: flexible ones introduced during type inference, and rigid ones bound by $\forall$-quantifiers, we have two kinds of substitution operation depending on which type variables we wish to replace. These are called `substFlex` and `substRigid` in the `Subst.hs` Haskell module, substituting for flexible and rigid type variables, respectively.

5 Type Inference Rules

The type inference are rules presented in Figure 3 using the judgement form:

$$\boxed{\Gamma_1 \vdash e \overset{\text{INF}}{\Longrightarrow} \tau \dashv \Gamma_2}$$

where blue components on the left of $\overset{\text{INF}}{\Longrightarrow}$ can be viewed as *inputs* to the algorithm, while red components on the right of $\overset{\text{INF}}{\Longrightarrow}$ can be viewed as outputs. The type inference rules directly encode an algorithm with explicit inputs and outputs. Inputs to the conclusion are inputs to the first premiss (by convention the left-most premiss of a rule). The outputs of a premiss can be used as inputs to subsequent premisses. The left-hand typing environment and expression are inputs to any rule. The right-hand typing environment and the type are outputs. The judgement encodes a substitution from type variables in the input typing environment to types in the output typing environment.

Such an algorithm has type signature:

```
inferExp :: Gamma -> Exp -> TC (Type, Gamma)
```

The goal is to *infer* the type of the top-level binding by inferring the types for all its sub-expressions, and propagating this information upwards through the program source tree. You should replace the unannotated top-level binding (*i.e.* `main`) with an explicitly typed equivalent.

5.1 Contexts for Polymorphism, Contexts for Type Inference

To support polymorphism and enable type inference, the grammar for typing contexts Γ is extended to track *unification* or *flexible* type variables, and their *instantiation* with a concrete type, if any. They are "flexible" in the sense that they are introduced by the type inference algorithm in order to establish constraints between types.

A context tracks both *declarations* and *definitions* of flexible type variables. A declaration is denoted by α in this document, or α `:=HOLE` in Haskell[2] and declares α to be the name of a flexible type variable which has not yet been instantiated with a concrete type. On the other hand, a definition is denoted by $\alpha := \tau$ for some type variable name α and type τ. The full grammar[3] for contexts looks like this:

$$\Gamma \quad ::= \quad \epsilon \mid \Gamma \cdot x : \sigma \mid \Gamma \cdot \alpha \mid \Gamma \cdot \alpha := \tau \mid \Gamma \cdot \bullet$$

[2] Again, Table 1 summaries these conventions.

[3] The $\bullet$ is explained later.

We distinguish a context from a *suffix* Δ which is a list containing only type variable declarations or definitions:

$$\Delta \quad ::= \quad [] \mid \alpha :: \Delta \mid \alpha := \tau :: \Delta$$

5.2 Generalisation

Generalisation is the process of introducing zero or more $\forall$ quantified type variables on a type to form a *type scheme*. This process must be minimal in the sense that it generalises only those type variables relevant for the current type inference problem. The judgement form is:

$$\boxed{\Gamma_1 \vdash e \overset{\text{GEN}}{\Longrightarrow} \sigma \dashv \Gamma_2}$$

where from input context Γ_1 and expression e the algorithm infers the *type scheme* σ and output context Γ_2. There is only one rule for generalisation given by:

$$\textsc{Gen}\ \frac{\Gamma_1 \cdot \bullet \vdash e \overset{\text{INF}}{\Longrightarrow} \tau \dashv \Gamma_2 \cdot \bullet \cdot \Delta \quad \sigma = \text{gen}(\text{rev}(\Delta), \tau, [])}{\Gamma_1 \vdash e \overset{\text{GEN}}{\Longrightarrow} \sigma \dashv \Gamma_2}$$

where $\text{rev}(\Delta)$ reverses the elements in the suffix Δ [4].

The algorithm keeps track of relevant type variables by using *markers*, denoted by $\bullet$, which split the context up into localities. A type variable occurring to the right of a marker can be safely generalised in a type scheme because the surrounding context does not depend on it. In the GEN rule, we can generalise τ over Δ (which contains only type variable declarations and definitions) to form the type scheme σ:

$$\begin{array}{lcl} \text{gen}([], \tau, \overline{\beta}) & \triangleq & \texttt{forall}\ \overline{\beta}.\, \tau \\ \text{gen}(\alpha :: \Delta, \tau, \overline{\beta}) & \triangleq & \text{gen}(\Delta, \tau[\alpha^{\text{F}} := \alpha^{\text{R}}], \alpha :: \overline{\beta}) \\ \text{gen}(\alpha := \tau' :: \Delta, \tau, \overline{\beta}) & \triangleq & \text{gen}(\Delta, \tau[\alpha^{\text{F}} := \tau'], \overline{\beta}) \end{array}$$

Type variables can gain more global relevance by moving their declarations to the left of markers. Such a movement is irreversible, once a type variable has moved beyond a marker it can never be moved back "to the right".

To keep type inference tractable, generalisation is performed only on let-bindings and the top-level `main` binding. Recursive functions are not generalised. Pay close attention to the premisses of the LET and PROGRAM typing rules in Figure 3 to see where generalisation should be used.

5.3 Unification

Solving type inference problems depends on solving *unification* problems between types. Just like the type inference rules, unification can be expressed algorithmically as a transformation on the typing context. Some of these rules were discussed in lectures and are repeated in Figure 1 (except a few structural and symmetric rules — fill in the details!) and take the following form:

$$\boxed{\Gamma_1 \vdash \tau_1 \sim \tau_2 \overset{\text{UNI}}{\Longrightarrow} \Gamma_2}$$

where, again, inputs are blue and outputs are red.

5.4 Instantiation

The unification judgement depends on *instantiation*, that is, solving an equation involving a flexible type variable (unification variable) and any type which is not a flexible type variable. This judgement (Figure 2) takes the following form:

$$\boxed{\Gamma_1 \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2}$$

[4] Reversing Δ is required by our definition of $\text{gen}(-,-,-)$ because we need to process more local type variables first in case they depend on later type variables appearing in Δ. You can avoid the reversal if you first build a simultaneous substitution (Section 6) out of Δ.

$$\boxed{\Gamma_1 \vdash \tau_1 \sim \tau_2 \overset{\text{UNI}}{\Longrightarrow} \Gamma_2}$$

$$\textsc{Refl}\ \frac{}{\Gamma_1 \vdash \alpha^{\text{F}} \sim \alpha^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_1} \qquad \textsc{Arrow}\ \frac{\Gamma_1 \vdash \tau_1 \sim \tau_1' \overset{\text{UNI}}{\Longrightarrow} \Gamma_2 \quad \Gamma_2 \vdash \tau_2 \sim \tau_2' \overset{\text{UNI}}{\Longrightarrow} \Gamma_3}{\Gamma_1 \vdash \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2' \overset{\text{UNI}}{\Longrightarrow} \Gamma_3}$$

$$\textsc{Inst}\ \frac{\tau \text{ not a flexvar} \quad \Gamma_1 \mid [] \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \vdash \alpha^{\text{F}} \sim \tau \overset{\text{UNI}}{\Longrightarrow} \Gamma_2} \qquad \textsc{Defn}\ \frac{\alpha \neq \beta}{\Gamma_1 \cdot \alpha \vdash \alpha^{\text{F}} \sim \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_1 \cdot \alpha := \beta^{\text{F}}}$$

$$\textsc{Subst}\ \frac{\Gamma_1 \vdash \alpha^{\text{F}}[\rho := \tau] \sim \beta^{\text{F}}[\rho := \tau] \overset{\text{UNI}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot \rho := \tau \vdash \alpha^{\text{F}} \sim \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_2 \cdot \rho := \tau} \qquad \textsc{Skip-Ty}\ \frac{\rho \neq \alpha \quad \rho \neq \beta \quad \Gamma_1 \vdash \alpha^{\text{F}} \sim \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot \rho \vdash \alpha^{\text{F}} \sim \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_2 \cdot \rho}$$

$$\textsc{Skip-Mark}\ \frac{\Gamma_1 \vdash \alpha^{\text{F}} \sim \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot \bullet \vdash \alpha^{\text{F}} \sim \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_2 \cdot \bullet} \qquad \textsc{Skip-Tm}\ \frac{\Gamma_1 \vdash \alpha^{\text{F}} \sim \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot x : \sigma \vdash \alpha^{\text{F}} \sim \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_2 \cdot x : \sigma}$$

Figure 1: Unification Rules

$$\boxed{\Gamma_1 \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2}$$

$$\textsc{Defn}\ \frac{\alpha \notin \texttt{FTV}\,(\tau)}{\Gamma_1 \cdot \alpha \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_1 \cdot \Delta \cdot \alpha := \tau} \qquad \textsc{Depend}\ \frac{\beta \neq \alpha \quad \beta \in \texttt{FTV}\,(\tau) \quad \Gamma_1 \mid \beta :: \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot \beta \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2}$$

$$\textsc{Subst}\ \frac{\Gamma_1 \cdot \Delta \vdash \alpha^{\text{F}}[\beta^{\text{F}} := \tau'] \sim \tau[\beta^{\text{F}} := \tau'] \overset{\text{UNI}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot \beta := \tau' \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2 \cdot \beta := \tau'} \qquad \textsc{Skip-Ty}\ \frac{\beta \neq \alpha \quad \beta \notin \texttt{FTV}\,(\tau) \quad \Gamma_1 \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot \beta \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2 \cdot \beta}$$

$$\textsc{Skip-Tm}\ \frac{\Gamma_1 \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot x : \sigma \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2 \cdot x : \sigma} \qquad \textsc{Skip-Mark}\ \frac{\Gamma_1 \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2}{\Gamma_1 \cdot \bullet \mid \Delta \vdash \alpha \sim \tau \overset{\text{INST}}{\Longrightarrow} \Gamma_2 \cdot \bullet}$$

Figure 2: Algorithmic instantiation rules

The algorithm simplifies the problem by moving through the context, similar to the strategy employed during unification for the case of unifying two flexible type variables. Some points worth noting:

- $\texttt{FTV}\,(\tau)$ computes the set of names of flexible type variables occurring in τ;
- For DEFN, we must check that the no flexible type variables occurring in τ have the name α. This condition is known as the *occurs check* and prevents cyclic dependencies which are unsound;
- Instantiation accumulates a list Δ recording the type variable dependencies of τ which must be given more global relevance in order to solve for α;
- For DEFN, the dependencies of τ are placed to the right of the definition of α because by setting $\alpha{:=}\tau$, it also depends on type variables in Δ;

$\boxed{\Gamma_1 \vdash e \overset{\text{INF}}{\Longrightarrow} \tau \dashv \Gamma_2}$

VAR
$$\frac{x : \forall\, \overline{\alpha_i}.\, \tau \in \Gamma}{\Gamma \vdash x \overset{\text{INF}}{\Longrightarrow} \tau[\overline{\alpha^{\text{R}}{}_i := \beta^{\text{F}}{}_i}] \dashv \Gamma \cdot \overline{\beta_i}}\ \beta_i \textit{ fresh}$$

INT
$$\frac{n \text{ an integer}}{\Gamma \vdash n \overset{\text{INF}}{\Longrightarrow} \texttt{Int} \dashv \Gamma}$$

CON
$$\frac{\text{conType}(k) = \forall\, \overline{\alpha_i}.\, \tau}{\Gamma \vdash \texttt{Con}\ k \overset{\text{INF}}{\Longrightarrow} \tau[\overline{\alpha^{\text{R}}{}_i := \beta^{\text{F}}{}_i}] \dashv \Gamma \cdot \overline{\beta_i}}\ \beta_i \textit{ fresh}$$

PRIM
$$\frac{\text{primOpType}(o) = \forall\, \overline{\alpha_i}.\, \tau}{\Gamma \vdash \texttt{Prim}\ o \overset{\text{INF}}{\Longrightarrow} \tau[\overline{\alpha^{\text{R}}{}_i := \beta^{\text{F}}{}_i}] \dashv \Gamma \cdot \overline{\beta_i}}\ \beta_i \textit{ fresh}$$

APP
$$\frac{\Gamma_1 \vdash e_1 \overset{\text{INF}}{\Longrightarrow} \tau_1 \dashv \Gamma_2 \quad \Gamma_2 \vdash e_2 \overset{\text{INF}}{\Longrightarrow} \tau_2 \dashv \Gamma_3 \quad \Gamma_3 \cdot \alpha \vdash \tau_1 \sim \tau_2 \to \alpha^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_4}{\Gamma_1 \vdash \texttt{Apply}\ e_1\ e_2 \overset{\text{INF}}{\Longrightarrow} \alpha^{\text{F}} \dashv \Gamma_4}\ \alpha \textit{ fresh}$$

IF-THEN-ELSE
$$\frac{\begin{array}{c}\Gamma_1 \vdash e \overset{\text{INF}}{\Longrightarrow} \tau \dashv \Gamma_2 \quad \Gamma_2 \vdash \tau \sim \texttt{Bool} \overset{\text{UNI}}{\Longrightarrow} \Gamma_3 \\ \Gamma_3 \vdash e_t \overset{\text{INF}}{\Longrightarrow} \tau_t \dashv \Gamma_4 \quad \Gamma_4 \vdash e_f \overset{\text{INF}}{\Longrightarrow} \tau_f \dashv \Gamma_5 \quad \Gamma_5 \vdash \tau_t \sim \tau_f \overset{\text{UNI}}{\Longrightarrow} \Gamma_6\end{array}}{\Gamma_1 \vdash \texttt{If}\ e\ e_t\ e_f \overset{\text{INF}}{\Longrightarrow} \tau_t \dashv \Gamma_6}$$

CASE
$$\frac{\begin{array}{c}\Gamma_1 \vdash e \overset{\text{INF}}{\Longrightarrow} \tau \dashv \Gamma_2 \quad \Gamma_2 \cdot \alpha \cdot \beta \vdash \tau \sim \alpha^{\text{F}} + \beta^{\text{F}} \overset{\text{UNI}}{\Longrightarrow} \Gamma_3 \\ \Gamma_3 \cdot x : \alpha^{\text{F}} \vdash e_1 \overset{\text{INF}}{\Longrightarrow} \tau_1 \dashv \Gamma_4 \cdot x : \alpha^{\text{F}} \cdot \Delta \quad \Gamma_4 \cdot \Delta \cdot y : \beta^{\text{F}} \vdash e_2 \overset{\text{INF}}{\Longrightarrow} \tau_2 \dashv \Gamma_5 \cdot y : \beta^{\text{F}} \cdot \Delta' \\ \Gamma_5 \cdot \Delta' \vdash \tau_1 \sim \tau_2 \overset{\text{UNI}}{\Longrightarrow} \Gamma_6\end{array}}{\Gamma_1 \vdash \texttt{Case}\ e\ \ x.e_1\ \ y.e_2 \overset{\text{INF}}{\Longrightarrow} \tau_1 \dashv \Gamma_6}\ \alpha, \beta \textit{ fresh}$$

RECFUN
$$\frac{\Gamma_1 \cdot \alpha \cdot \beta \cdot x : \alpha^{\text{F}} \cdot f : \beta^{\text{F}} \vdash e \overset{\text{INF}}{\Longrightarrow} \tau \dashv \Gamma_2 \cdot x : \alpha^{\text{F}} \cdot f : \beta^{\text{F}} \cdot \Delta \quad \Gamma_2 \cdot \Delta \vdash \beta^{\text{F}} \sim \alpha^{\text{F}} \to \tau \overset{\text{UNI}}{\Longrightarrow} \Gamma_3}{\Gamma_1 \vdash \texttt{Recfun}\ f.x.e \overset{\text{INF}}{\Longrightarrow} \beta^{\text{F}} \dashv \Gamma_3}\ \alpha, \beta \textit{ fresh}$$

LET
$$\frac{\Gamma_1 \vdash e_1 \overset{\text{GEN}}{\Longrightarrow} \sigma \dashv \Gamma_2 \quad \Gamma_2 \cdot x : \sigma \vdash e_2 \overset{\text{INF}}{\Longrightarrow} \tau \dashv \Gamma_3 \cdot x : \sigma \cdot \Delta}{\Gamma_1 \vdash \texttt{Let}\ e_1\ \ x.e_2 \overset{\text{INF}}{\Longrightarrow} \tau \dashv \Gamma_3 \cdot \Delta}$$

$\boxed{\Gamma_1 \vdash p \overset{\text{INF\_PROG}}{\Longrightarrow} \sigma \dashv \Gamma_2}$

PROGRAM
$$\frac{\Gamma_1 \vdash e \overset{\text{GEN}}{\Longrightarrow} \sigma \dashv \Gamma_2}{\Gamma_1 \vdash \texttt{Program}\ e \overset{\text{INF\_PROG}}{\Longrightarrow} \sigma \dashv \Gamma_2}$$

Figure 3: Algorithmic Type Inference Rules for `MinHS` expressions and programs

6 Substitution

Substitutions are implemented as an abstract data type, defined in `Subst.hs`. `Subst` is an instance of the `Monoid` type class, which is defined in the standard library as follows:

```
class Monoid a where
  mappend :: a -> a -> a   -- also written as the infix operator <>
  mempty  :: a
```

For the `Subst` instance, `mempty` corresponds to the empty substitution, and `mappend` is *substitution composition*. That is, applying the substitution `a <> b` is the same as applying `a` and `b` simultaneously. It should be reasonably clear that this instance obeys the *monoid laws*:

```
mempty <> x == x               -- left identity
x <> mempty == x               -- right identity
x <> (y <> z) == (x <> y) <> z -- associativity
```

It is also *commutative* (`x <> y == y <> x`) assuming that the substitutions are *disjoint* (i.e that $dom(x) \cap dom(y) = \emptyset$). In the type inference algorithm, your substitutions are all applied in order and thus should be disjoint, therefore this property should hold.

You can use this <> operator to combine multiple substitutions into a single substitution; however, there should be little need to use anything more complicated than single substitutions for this algorithm.

You can construct a singleton substitution, which replaces one variable, with the `=:` operator, so the substitution `("a" =: FlexVar "b") <> ("b" =: FlexVar "c")` using `substFlex` is a substitution which renames all flexible type variables with name a or b with the name c.

The `Subst` module also includes a variety of functions for running substitutions on types and expressions for both rigid and flexible type variable replacement. In particular, the module provides two functions for expressions and types respectively, for substitution of type variables:

```
-- | Perform substitution on rigid type variables.
substRigid :: Subst -> Type -> Type
-- | Perform substitution on flexible type variables.
substFlex  :: Subst -> Type -> Type
```

and similar for `Exp`. You can use these functions for implementing some of the algorithimic rules.

7 Errors and Fresh Names

Thus far, the following type signature would be sufficient for implementing our type inference function:

```
inferExp :: Gamma -> Exp -> (Type, Gamma)
```

Unification is a partial function, however, so we want a principled way to handle the error cases, rather than just bail out with `error` calls.

To achieve this, we'll adjust the basic, pure signature for type inference to include the possibility of a `TypeError`:

```
inferExp :: Gamma -> Exp -> Either TypeError (Type, Gamma)
```

Even this, though, is not sufficient, as we cannot generate fresh, unique type variable names for use as flexible/unification variables:

```
freshId :: Id -- it is impossible for fresh to return a different
freshId = ?   -- value each time!
```

To solve this problem, we could pass an (infinite) list of unique names around our program, and `fresh` could simply take a name from the top of the list, and return a new list with the name removed:

```
fresh :: [Id] -> ([Id],Id)
fresh (x:xs) = (xs,x)
```

This is quite awkward though, as now we have to manually thread our list of identifiers throughout the entire inference algorithm:

```
inferExp :: Gamma -> Exp -> [Id] -> Either TypeError ([Id], (Type, Gamma))
```

To resolve this, we bundle both the `[Id]` state transformer and the `Either TypeError x` error handling into one abstract type, called `TC` (defined in `TCMonad.hs`)

```
newtype TC a = TC ([Id] -> Either TypeError ([Id], a))
```

One can think of `TC a` abstractly as referring to a *stateful action* that will, if executed, return a value of type `a`, or throw an exception.

As the name of the module implies, `TC` is a *Monad*, meaning that it exposes two functions (`return` and `>>=`) as part of its interface.

```
return :: a -> TC a
return = ...
(>>)  :: TC a -> TC b -> TC b
a >> b = a >>= const b
(>>=)  :: TC a -> (a -> TC b) -> TC b
a >>= b = ...
```

The function `return` is, despite its name, just an ordinary function which *lifts* pure values into a `TC` action that returns that value. The function `(>>)` (read *then*), is a kind of composition operator, which produces a `TC` action which runs two `TC` actions in sequence, one after the other, returning the value of the last one to be executed. Lastly, the function `(>>=)`, more general than `(>>)`, allows the second executed action to be determined by the return value of the first.

The `TCMonad.hs` module also includes a few built-in actions:

```
typeError :: TypeError -> TC a -- throw an error
freshId   :: TC Id             -- return a fresh type variable name
```

Haskell includes special syntactic sugar for monads, which allow for programming in a somewhat imperative style. Called `do` notation, it is simple sugar for `(>>)` and `(>>=)`.

```
do e                 --       e
do e; v              --       e >> do v
do p <- e; v         --       e >>= \p -> do v
do let x = y; v      --       let x = y in do v
```

This lets us write unification and type inference quite naturally. A simple example of the use of the `TC` monad is already provided to you, the `specialise` function, which takes a type with some number of quantifiers, and replaces all quantifiers with fresh variables (very useful in the type inference cases for variables, constructors, and primops):

```
specialise :: Scheme -> TC (Type, Suffix)
specialise (Forall xs t) =
  do ids <- freshForall xs
     return (S.substRigid (S.fromList (map (second FlexVar) ids)) t
            , map (flip (,) HOLE . snd) ids)
```

To run a `TC` action in your top level `infer` function, the `runTC` function can be used:

```
runTC :: TC a -> Either TypeError a
```

Please note: This function runs the TC action with the same source of fresh names each time! Using it more than once in your program is not likely to give correct results.

8 Program structure

A program in `MinHS` may evaluate to any non-function type, including an aggregate type. This is a valid `MinHS` program:

```
main = (1,(InL True, False));
```

which can be elaborated to the following type:

```
main :: forall 't. (Int * ((Bool + 't) * Bool))
     = (1,(InL True,False));
```

8.1 Type information

The most significant change to the language of assignment 1 is that the parser now accepts programs without any type information. Type declarations are not compulsory! Unless you are attempting the bonus parts of the assignment, you can assume that *no* type information will be provided in the program.

You can view the type information after your pass using `--dump type-infer`.

9 Implementing Type Inference

You are required to implement the function `infer`. Some stub code has been provided for you, along with some type declarations, and the type signatures of useful functions you may wish to implement. You may change any part of `TyInfer.hs` you wish, as long as it still provides the function `infer`, of the correct type. The stub code is provided only as a hint, you are free to ignore it.

10 Testing

Your assignments will be autotested rigorously. You are encouraged to autotest yourself. `MinHS` comes with a tester script, and you can add your own tests to this. Your assignment will be tested by comparing the output of `tyinf --dump type-infer` against the expected abstract syntax. Your solution must be α-equivalent to the expected solution.

Much like the previous assignment, you are given a suite of tests for Task 1. **Unlike the previous assignment**, the tests do not specify the expected results—note the lack of `.out` files. **Beware**: unless you add `.out` specifying the expected result yourself, the tests will report success no matter what you return.

It is up to you to write your own tests for the extension tasks.

In this assignment we make no use of the later phases of the compiler.

11 Building MinHS

Building MinHS is exactly the same as in Assignment 1.

To run the type inference pass and inspect its results, for `cabal` users type:

```
$ cabal run tyinf -- --dump type-infer foo.mhs
```

Users of `stack` should type:

```
$ stack exec tyinf -- --dump type-infer foo.mhs
```

You may wish to experiment with some of the debugging options to see, for example, how your program is parsed, and what abstract syntax is generated. Many `--dump` flags are provided, which let you see the abstract syntax at various stages in the compiler.

12 Late Penalty

You may submit up to five days (120 hours) late. However, **no late penalty will be applied** unless you specifically request it. I can't imagine why you would request it, but if you do, each day of lateness corresponds to a 5% reduction of your total mark. For example, if your assignment is worth 88% and you submit it two days late, you get 78%.

If you submit it more than five days late, you get 0%, regardless of whether you request a late penalty or not.

Course staff cannot grant assignment extensions—if you need an extension, you have to apply for special consideration through the standard procedure. More information here: `https://www.student.unsw.edu.au/special-consideration`

13 Plagiarism

Many students do not appear to understand what is regarded as plagiarism. This is no defense. Before submitting any work you should read and understand the UNSW plagiarism policy `https://student.unsw.edu.au/plagiarism`.

All work submitted for assessment must be entirely your own work. We regard unacknowledged copying of material, in whole or part, as an extremely serious offence. In this course submission of any work derived from another person, or solely or jointly written by and or with someone else, without clear and explicit acknowledgement, will be severely punished and may result in automatic failure for the course and a mark of zero for the course. Note this includes including unreferenced work from books, the internet, etc.

Do not provide or show your assessable work to any other person. Allowing another student to copy from you will, at the very least, result in zero for that assessment. If you knowingly provide or show your assessment work to another person for any reason, and work derived from it is subsequently submitted you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep your work private until submissions have closed.

If you are unsure about whether certain activities would constitute plagiarism ask us before engaging in them!

References

[1] *Haskell 2010 Language Report*, editor Simon Marlow, (2010) `http://www.haskell.org/onlinereport/Haskell2010`

[2] Robert Harper, *Programming Languages: Theory and Practice*, (Draft of 19 Sep 2005), `https://people.cs.uchicago.edu/~blume/classes/aut2008/proglang/text/offline.pdf`.