Functions The rules for function expressions are all similar, though with a few differences between them. All of them create a fresh type variable X_1 to represent the type of their argument, and the resulting type is always $X_1 \to T_1$, where T_1 is the type of the body of the function.

When calling the collection algorithm on the body of the function (i.e. e), the typing environment Γ is modified by addind a new association between the identifier x and the type X_1.

$$\frac{\Gamma \vdash fresh(X) = X_1 \qquad \{x \to X_1\} \cup \Gamma \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash \texttt{fn}\ x \Rightarrow e : X_1 \to T_1 \mid C_1 \mid \gamma_1} \quad \text{(T-Fn)}$$

Recursive functions add the same association between x and X_1, but they also create a new association for the name of the function, f. If the function is implicitly typed, a new type variable, X_2, is used to represent the type of the function. Thus, f is associated to X_2, and a new constraint between X_2 and $X_1 \to T_1$ is created.

$$\frac{\begin{array}{c}\Gamma \vdash fresh(X) = X_1 \qquad \Gamma \vdash fresh(X) = X_2 \\ \{f \to X_2, x \to X_1\} \cup \Gamma \vdash e : T_1 | C_1 \mid \gamma_1\end{array}}{\Gamma \vdash \texttt{rec}\ f\ x \Rightarrow e : X_1 \to T_1 \mid C_1 \cup \{X_2 = X_1 \to T_1\} \mid \gamma_1} \quad \text{(T-Rec)}$$

If the function is explicitly typed, however, no new type variables are created. Instead, f is associated directly to $X_1 \to T$, and a constraint to guarantee that the provided type is correct is created (that is, that T is equal to T_1).

$$\frac{\Gamma \vdash fresh(X) = X_1 \qquad \{f \to (X_1 \to T, x \to X_1)\} \cup \Gamma \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash \texttt{rec}\ f : T\ \ x \Rightarrow e : X \to T_1 \mid C_1 \cup \{T_1 = T\} \mid \gamma_1} \quad \text{(T-Rec2)}$$

Built-in Functions None of the built-in functions create any constraints or unification environment, nor do their types depend on a typing environment. Some functions, because of their polymorphic nature, require creation of fresh type variables.

Numerical Functions These functions all manipulate Int values, with negate ($-$) being the only function with a single argument.

$$\Gamma \vdash + : \text{Int} \to \text{Int} \to \text{Int} \quad \text{(T-+)}$$

$$\Gamma \vdash - : \text{Int} \to \text{Int} \to \text{Int} \quad \text{(T–)}$$

$$\Gamma \vdash * : \text{Int} \to \text{Int} \to \text{Int} \quad \text{(T-*)}$$

$$\Gamma \vdash \div : \text{Int} \to \text{Int} \to \text{Int} \quad \text{(T-÷)}$$

$$\Gamma \vdash - : \text{Int} \to \text{Int} \quad \text{(T-Negate)}$$

Equality Functions These functions do not require a specific type for their arguments, but they both must be equal and conform to the *Equatable* trait.

$$\frac{\Gamma \vdash fresh(X^{\{Equatable\}}) = T}{\Gamma \vdash =: T \to T \to \text{Bool}} \quad (\text{T-=})$$

$$\frac{\Gamma \vdash fresh(X^{\{Equatable\}}) = T}{\Gamma \vdash \neq: T \to T \to \text{Bool}} \quad (\text{T-}\neq)$$

Inequality Functions Similar to equality, both arguments must have the same type and conform to the *Orderable* trait.

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash <: T \to T \to \text{Bool}} \quad (\text{T-}<)$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash \leq: T \to T \to \text{Bool}} \quad (\text{T-}\leq)$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash >: T \to T \to \text{Bool}} \quad (\text{T-}>)$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash \geq: T \to T \to \text{Bool}} \quad (\text{T-}\geq)$$

Boolean Functions Both functions require two Bool arguments, returning another Bool.

$$\Gamma \vdash \vee : \text{Bool} \to \text{Bool} \to \text{Bool} \quad (\text{T-}\vee)$$

$$\Gamma \vdash \wedge : \text{Bool} \to \text{Bool} \to \text{Bool} \quad (\text{T-}\wedge)$$

Accessor Functions These functions manipulate accessors, creating fresh type variables to represent all necessary types.

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2}{\Gamma \vdash \text{get} : T_2\#T_1 \to T_2 \to T_1} \quad (\text{T-}\textsc{get})$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2}{\Gamma \vdash \text{set} : T_2\#T_1 \to T_1 \to T_2 \to T_2} \quad (\text{T-}\textsc{set})$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2 \qquad \Gamma \vdash fresh(X) = T_3}{\Gamma \vdash \text{stack} : T_2\#T_1 \to T_1\#T_3 \to T_2\#T_3} \quad (\text{T-}\textsc{stack})$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2 \qquad \Gamma \vdash fresh(X) = T_3}{\Gamma \vdash \text{distort} : T_2\#T_1 \to (T_1 \to T_3) \to (T_3 \to T_1 \to T_1) \to T_2\#T_3} \quad (\text{T-}\textsc{distort})$$

Constructors The rule for typing constructors is very simple. The type is extracted from the environment, and then a fresh instance is generated from that type.

$$\frac{\Gamma(con) = T \qquad \Gamma \vdash fresh(T) = T'}{\Gamma \vdash con : T'} \tag{T-Con}$$

Application The constraint collection rule for an application is simple, creating just one fresh type variable and one new constraint. The type variable X_1, represents the type of the result of the application, and, therefore, is the return type of the collection. Furthermore, the type of e_1, T_1, must be equal to a function that takes T_2 (the type of e_2) as an argument and returns X_1.

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \mid \gamma_2 \qquad \Gamma \vdash fresh(X) = X_1}{\Gamma \vdash e_1\ e_2 : X_1 \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to X_1\} \mid \gamma_1 \cup \gamma_2} \tag{T-App}$$

Identifiers The type of an identifier is, like for constructors, completely defined by its typing association in the environment. The typing rule does not create a fresh instance of this type, since the environment already does this when returning types that are universally bound.

$$\frac{\Gamma\,(x) = T}{\Gamma \vdash x : T} \tag{T-Ident}$$

Records The constraint collection rule for a record is relatively straightforward. Each field of the record is passed through the collection algorithm, and the resulting types are combined into a single record type with their matching labels. Similarly, the resulting constraints and unification environments are combined by union.

$$\frac{\forall\, k \in [1, n] \quad \Gamma \vdash e_k : T_k \mid C_k \mid \gamma_k}{\Gamma \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} : \{l_1 : T_1,\ \ldots\ l_n : T_n\} \mid \bigcup_{i=1}^{n} C_i \mid \bigcup_{i=1}^{n} \gamma_i} \tag{T-Record}$$

Accessors The constraint rules for simple label accessors relies on type variables and record label traits. A new fresh type variable, T_1, is generated, representing the type of the field being accessed. Another new fresh type variable, T_2, which must conform to the record label trait associating the label l to the type T_1, is generated, representing the type of the record that is being accessed.

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X^{\{l:T_1\}}) = T_2}{\Gamma \vdash \#l : T_2\#T_1} \tag{T-Label}$$

A joined accessor does not use record label traits, but instead relies on type variables and constraints to guarantee the correct type information.

A single type variable X_0, represents the record being accessed. For every component e_i of the accessor, a new type variable X_i is generated, along with the resulting type T_i of calling the constraint collection algorithm. The type T_i is then constrained to be equal to $X_0\#X_i$, indicating that all components refer the same record, but access fields with (possibly) different types.

Finally, the resulting type is an accessor that returns a tuple composed of all X_i when accessing a record of type X_0.

$$\frac{fresh(X) = X_0 \qquad \forall\ i \in [1,n] \quad \Gamma \vdash fresh(X) = X_i \land \Gamma \vdash e_i : T_i \mid C_i \mid \gamma_i}{\Gamma \vdash \#(e_1,\ \ldots\ e_n) : X_0\#(X_i,\ \ldots\ X_n) \mid \bigcup_{i=1}^{n} C_i \cup \{T_i = X_0\#X_i\} \mid \bigcup_{i=1}^{n} \gamma_i} \text{(T-Joined)}$$

Let Expression The constraint collection rule for a `let` expression depends on both the unification and the application algorithms.

The expression e_1 is passed through the constraint collection algorithm, resulting in a type T_1, a set of constraints C_1 and a unification environment γ_1. The constraints C_1 are then unified (see **??**) under the unification environment γ_1, resulting in a substitution σ.

The substitution σ is then applied (see **??**) to the type T_1, resulting in a principle type T_1'. The substitution is also applied to the environment Γ, and the result of this application is used to evaluate a universal match between p and T_1', resulting in a new set of constraints C_1' and a new typing environment Γ'.

Finally, the type of the expression e_2 is obtained under the environment Γ'.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \qquad \gamma_1 \vdash \mathcal{U}(C_1) = \sigma \qquad \sigma(T_1) = T_1' \\ \sigma(\Gamma) \vdash match_U(p, T_1') = C_1', \Gamma' \qquad \Gamma' \vdash e_2 : T_2 \mid C_2 \mid \gamma_2\end{array}}{\Gamma \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 : T_2 \mid C_1' \cup C_1 \cup C_2 \mid \gamma_2 \cup \gamma_1} \text{(T-Let)}$$

Match Expression The constraint collection rule for a match expression requires an auxiliary function, much like its operational semantic rule. A fresh type variable X_1 is created, representing the output type of the expression and, along with the type T of the expression e, is used to validate every $match_i$ in the expression.

$$\frac{\begin{array}{c}\Gamma \vdash e : T \mid C \mid \gamma \qquad \Gamma \vdash fresh(X) = X_1 \\ \forall i \in [1..n]\, \Gamma \vdash validate(match_i, T, X_1) = C_i \mid \gamma_i\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ match_1, ...\ match_n : X_1 \mid C \cup \bigcup_{i=1}^{n} C_i \mid \gamma \cup \bigcup_{i=1}^{n} \gamma_i} \text{(T-Match)}$$

The *validate* function takes a *match* expression, a type T_{in}, representing the type of the pattern, and a T_{out}, representing the result of evaluating the *match* expression, as inputs. The function outputs a set of constraints and a unification environment if successful.

For an unconditional *match*, the pattern in matched against the provided input type T_{in} and the type of the expression e is constrained to equal the provided type T_{out}. It is important to realize that the typing environment returned by the match (i.e. Γ') is used only to obtain the type of e, since any identifiers bound in the pattern p can only be used inside a single *match* expression.

$$\frac{\Gamma \vdash match(p, T_{in}) = C, \Gamma' \qquad \Gamma' \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash validate(p \rightarrow e, T_{in}, T_{out}) = C \cup C_1 \cup \{T_1 = T_{out}\} \mid \gamma_1}$$

The same holds true for a conditional *match*, with the added verification that the type of e_1 must be equal to *Bool*.

$$\frac{\begin{array}{c}\Gamma \vdash match(p, T_{in}) = C, \Gamma' \qquad \Gamma' \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \\ \Gamma' \vdash e_2 : T_2 \mid C_2 \mid \gamma_2\end{array}}{\begin{array}{c}\Gamma \vdash validate(p \text{ when } e_1 \rightarrow e_2, T_{in}, T_{out}) = \\ C \cup C_1 \cup C_2 \cup \{T_1 = Bool, T_2 = T_{out}\} \mid \gamma_1 \cup \gamma_2\end{array}}$$

Exception The *raise* expression simply creates and returns a new fresh type variable.

$$\frac{\Gamma \vdash fresh(X) = X_1}{\Gamma \vdash raise : X_1} \tag{T-Raise}$$