0.1 Type System

Like many programming languages, *V* has a type system. A type system is a way of statically analyzing programs to decide whether they are well-formed or not. To do this, every expression in the abstract syntax tree has, associated with it, type information.

Typing rules are then used to check that a program is correctly constructed. If a program passes the type system, guarantees are made about its execution. *V*'s type system is not *secure* in the sense that correctly typed programs will always have correct execution.

Examples of programs that pass the type system but fail to execute are division by zero and accessing the head of an empty list. In general, however, these errors are caused by incomplete pattern matching on algebraic data types, and most errors are still caught by the type system.

V's type system is a Hindley-Milner style type system, with support for implicit type annotations and let polymorphism. Furthermore, *traits* allow ad-hoc polymorphism and, used with records, a kind of structural subtyping (more details will be provided later).

The Type Inference Algorithm Since *V* allows implicitly typed expressions (that is, expressions without any type annotations provided by the programmer), it is necessary to infer, and not only check, the type of expressions. A constraint-based inference system is used, which divides the algorithm into three parts: constraint collection, in which the abstract syntax tree is traversed and both a type and a list of type equality constraints is generated; constraint unification, in which the list of constraints is condensed into a type substitution; and substitution application, which applies the substitution to the type to obtain a principle type.

0.1.1 Constraint Collection

The first step of the type inference algorithm is the collection of type constraints. A type constraint is an equality between two types, and so the result of the constraint collection is a system of equations.

The constraint collection algorithm takes, as input, an expression e and a typing environment Γ (defined below), and produces, as an output, a type T, a set of constraints C and a unification environment γ (defined in 0.1.2). The algorithm is given as a set of rules of the form:

$$\Gamma \vdash e : T \mid C \mid \gamma \tag{T}$$

Constraints As described above, a set of constraints C is composed of equalities between two types.

$$C \quad ::= \quad \{\} \mid \{T_1 = T_2\} \cup C$$

Environment The type inference environment is a 2-tuple with the following components:

1. Mapping between constructors and their type

   Every constructor has a type associated with it. This type will be a function if the constructor has arity greater than 0, and can contain variable types.

2. Mapping between identifiers and type associations

   An identifier can be either simply or universally bound to a type. The difference between these associations will be explained later.

$$
\begin{array}{lll}
\Gamma & ::= & (constructors, vars) \\
 & & \\
constructors & ::= & \{\} \mid \{con \rightarrow T\} \cup constructors \\
vars & ::= & \{\} \mid \{x \rightarrow assoc\} \cup vars \\
 & & \\
assoc & ::= & T \qquad \text{(Simple Association)} \\
 & \mid & \forall X_1, \ldots X_n \,.\, T \qquad \text{(Universal Association)}
\end{array}
$$

Type Associations When identifiers are bound in a program (with pattern matching, for example), an association between the identifier and its type is added to the typing environment. Depending on the type of binding, however, this association can be one of two types: simple or universal.

A simple association binds a name to a monomorphic type. This type can "simple", such as Int, or it can contain variable types, such as $X \rightarrow \text{Bool}$. In either case, however, the type is "constant", and it is returned unchanged from the environment.

A universal association binds a name to a type scheme. A type scheme is composed of a type T and any number of type variables $X_1, \cdots X_n$. These variables are free in the type T, and a new instance of them is generated every time the association is returned from the environment.

Universal associations allow for polymorphic functions in the language, so each use of the function does not add constraints to other uses. The only expressions that create universal associations are let-expressions. This means that function parameters cannot be polymorphic, since function parameters are bound to simple type associations.

Free Variables Type variables can either be bound or free relative to an environment. Bound variable types are those that are associated to an identifier. This association must be a simple association, but the variable type can occur anywhere in the type tree. As an example, the type variable X_1 is bound in the environment below:

$$\Gamma = \{x \rightarrow (Int, X_1, Char)\}$$

Inversely, free type variables are all those that do not occur bound in the environment.

A helper function, $\Gamma \vdash free(T)$, returns the set of all free type variables in the type T. Another function, $\Gamma \vdash fresh(T)$, returns a type T' in which all free type variables in T are replaced by new, unbound type variables. Both of these functions require an environment Γ with which to judge whether type variables are free or not.

Pattern Matching One way to bind identifiers is by pattern matching. When a pattern is encountered (such as a `let` expression), it is necessary to match the type of the pattern with the type of the value.

To do this, two auxiliary *match* functions are defined. Both take, as input, a pattern p and a type T, returning a list of constraints and a modified typing environment.

The first of the functions, *match*, only creates simple type associations and is used in `match` expressions. The second, $match_U$, can create both simple and universal associations, being used in `let` expressions.

The following are the rules for the *match* function:

$$\Gamma \vdash match(x, T) = \{\}, \{x \to T\} \cup \Gamma$$

$$\Gamma \vdash match(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \to T\} \cup \Gamma$$

$$\Gamma \vdash match(\_, T) = \{\}, \Gamma$$

$$\Gamma \vdash match(\_ : T_{pat}, T) = \{T_{pat} = T\}, \Gamma$$

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash fresh(T') = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match(con\ p_1,\ \ldots\ p_n, T) = \{T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash fresh(T') = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match(con\ p_1,\ \ldots\ p_n : T_{pat}, T) = \{T_{pat} = T, T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1, n] \quad \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\}, T) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1, n] \quad \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\} : T_{pat}, T) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T, T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1, n] \quad \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i \qquad X_0^{\{\{l_i:X_i\},\ldots\{l_n:X_n\}\}}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}, T) = \{X_0 = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \quad \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i \qquad X_0^{\{\{l_i:X_i\},\ldots\{l_n:X_n\}\}}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\} : T_{pat}, T) = \{X_0 = T, T_{pat} = T\} \cup \bigcup_{i=1}^{n} c_i, \Gamma_n}$$

The rules for the $match_U$ are similar, but with a few key differences. Since universal matching is always called with a completely unified type in a `let` expression, certain concessions can be made. Furthermore, other differences arise due to the ability for the $match_U$ function to create universal type associations.

Because $match_U$ can create universal associations, it checks for any free variable types in the type T. If free variable types are found, then a universal association is made based on these variable types. If there are no free variable types, a simple association is created instead.

$$\frac{\Gamma \vdash free(T) = \{\}}{\Gamma \vdash match_U(x, T) = \{\}, \{x \to T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{X_1, \ldots, X_n\}}{\Gamma \vdash match_U(x, T) = \{\}, \{x \to \forall X_1, \ldots X_n \,.\, T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{\}}{\Gamma \vdash match_U(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \to T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{X_1, \ldots, X_n\}}{\Gamma \vdash match_U(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \to \forall X_1, \ldots X_n \,.\, T\} \cup \Gamma}$$

$$\Gamma \vdash match_U(\_, T) = \{\}, \Gamma$$

$$\Gamma \vdash match_U(\_ : T_{pat}, T) = \{T_{pat} = T\}, \Gamma$$

The next difference comes when matching against constructor patterns. Instead of creating a fresh instance of the type associated with the constructor by replacing all variable types with fresh variable types, the type T (passed as paramenter to $match_U$) is used.

This function is called $\Gamma \vdash rebase(T_1, T_2)$, creating a new instance of T_1 based on T_2. Informally, this means that both the type T_1 and T_2 are traversed simultaneously and, when a variable type is encountered in T_1, it is replaced by the equivalent type in T_2.

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash rebase(T', T) = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1,n] \quad \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match_U(con\ p_1,\ \ldots\ p_n, T) = \{T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash rebase(T', T) = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match_U(con\ p_1,\ \ldots\ p_n : T_{pat}, T) = \{T_{pat} = T, T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

Finally, pattern matching on records in also slightly different. For complete record patterns, we know that the type T is a record type with the necessary fields, and so the matching rule becomes much smaller.

$$\frac{\begin{array}{c}T = \{l_1 : T_1, \ldots, l_n : T_n\} \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i\end{array}}{match(\{l_1 : p_1, \ldots, l_n : p_n\}, T) = \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c}T = \{l_1 : T_1, \ldots, l_n : T_n\} \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i\end{array}}{match(\{l_1 : p_1, \ldots, l_n : p_n\} : T_{pat}, T) = \{T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

For partial record patterns, however, a little more care must be taken. Since there may be fewer fields in the pattern than the type, a search must be done to match the correct sub-patterns with the sub-types.

$$\frac{\begin{array}{c}T = \{l'_1 : T_1, \ldots, l'_k : T_k\} \\ k \geq n \qquad \forall\ i \in [1, n] \quad \exists\ j \in [1, k] \quad l_i = l'_j \wedge \Gamma_{i-1} \vdash match(p_i, T_j) = C_i, \Gamma_i\end{array}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}, T) = \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c}T = \{l'_1 : T_1, \ldots, l'_k : T_k\} \\ k \geq n \qquad \forall\ i \in [1, n] \quad \exists\ j \in [1, k] \quad l_i = l'_j \wedge \Gamma_{i-1} \vdash match(p_i, T_j) = C_i, \Gamma_i\end{array}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\} : T_{pat}, T) = \{T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

Constraint Collection Rules Every expression in V has a rule for constraint collection, similar to how every expression has a rule for its semantic evaluation.

If a rule does not create any constraints or unification environment (i.e. they are both empty), then these will be omited to improve readability. As an example, the following rule:

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

will be written as:

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

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.

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-IDENT)}$$

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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).

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

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 0.1.2) under the unification environment γ_1, resulting in a substitution σ.

The substitution σ is then applied (see 0.1.3) 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'.

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

$$\frac{\begin{array}{c}\Gamma \vdash e : T \mid C \mid \gamma_1 \qquad \Gamma \vdash fresh(X) = X_1 \\ \forall i \in [1..n]\, \Gamma \vdash validate(match_i, T, X_1) = C_i \mid \gamma_i\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ match_1, \ldots\ match_n : X_1 \mid C \cup \bigcup_{i=1}^{n} C_i} \text{ (T-Match)}$$

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

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

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

0.1.2 Unification

After constraint collection, the second step in the type inference algorithm is unification. Unification attempts to solve the set of equalities defined by the constraints collected in the previous step.

The algorithm takes, as input, a set of constraints C and a unification environment γ, and produces a substitution σ as output. The algorithm is given as a set of rules following the form:

$$\gamma \vdash \mathcal{U}(C) = \sigma \tag{U}$$

Unification Environment The unification environment is a set of trait specifications. Trait specifications are 3-tuples that define the requirements for a specific type to conform to a specific trait.

$$\gamma \quad ::= \quad \{\} \mid \{\text{trtSpec}\} \cup \gamma$$

$$\text{trtSpec} \quad ::= \quad (conT, Trait, [Traits_1, \ldots Traits_n]) \quad (n = \text{arity } conT)$$

A trait specification describes conformance to a trait. Some types, such as records and functions, have their trait conformance built into the language, and it is not necessary to use the unification environment to decide conformance. For a constructor type $conT\ T_1,\ \ldots\ T_n$ to conform to a trait $Trait$, however, the following criteria must hold:

1. There exists a trait specification $(conT, Trait, [Traits_1, \ldots Traits_n])$ in the unification environment
2. For all $i \in [1, n],\ T_i \in Traits_i$

This will be formally defined in the unification algorithm itself, but these are the general rules that govern trait conformance.

Substitution The result of applying the unification algorithm is a substitution σ. A substitution is a mapping from type variables to types.

$$\sigma \quad ::= \quad \{\} \mid \{X \to T\} \cup \sigma$$

0.1.3 Application

The last component of the type inference algorithm is the application of a type substitution. This replaces all type variables that are specified by the substitution, resulting in a new instance of the input type.

Application takes, as input, a type T and a substitution σ, producing another type T' as output. It is defined with rules of the form:

$$\sigma(T) = T' \tag{A}$$