V Syntax and Documentation

Arthur Giesel Vedana

August 1, 2017

Contents

1 Abstract Syntax Tree

1.1 Terms

e	::=	n	
	|	x	
	|	$(e_1,\ \ldots\ e_n)$	$(n \geq 2)$
	|	$\{l_1 : e_1,\ \ldots\ l_n : e_n\}$	$(n \geq 1)$
	|	$\#path$	
	|	$stack\ e_1\ e_2$	
	|	$distort\ e_1\ e_2\ e_3$	
	|	$get\ e_1\ e_2$	
	|	$set\ e_1\ e_2\ e_3$	
	|	$\texttt{let}\ p = e_1\ \texttt{in}\ e_2$	
	|	$\texttt{fn}\ p \Rightarrow e$	
	|	$e_1\ e_2$	
	|	$raise$	
	|	$\texttt{match}\ e\ \texttt{with}\ match_1, ...\ match_n$	$(n \geq 1)$
$path$	::=	l	
	|	$path_1\ .\ path_2$	
	|	$(e_1,\ \ldots\ e_n)$	$(n \geq 2)$
	|	$path\ [e_1,\ e_2]$	
$match$	::=	$p \rightarrow e$	
	|	$p\ \texttt{when}\ e_1 \rightarrow e_2$	
l	::=	$\{l_1, l_2, ...\}$	
p	::=	x	
	|	$\_$	
	|	n	
	|	$(p_1,\ \ldots\ p_n)$	$(n \geq 2)$
	|	$\{l_1 : p_1,\ \ldots\ l_n : p_n\}$	$(n \geq 1)$
	|	$\{l_1 : p_1,\ \ldots\ l_n : p_n, \ldots\}$	(partial record, $n \geq 1$)
x	::=	$\{x_0, x_1, \ldots\}$	
n	::=	$\mathbb{Z}$	

1.2 Types and Traits

$$
\begin{array}{lll}
T & ::= & \text{Int} \\
& | & T_1 \to T_2 \\
& | & (T_1, \ \ldots \ T_n) \qquad (n \geq 2) \\
& | & \{l_1 : T_1, \ \ldots \ l_n : T_n\} \qquad (n \geq 1) \\
& | & \#(T_1 \to T_2) \qquad \mathit{Accessor} \\
& | & X^{Traits}
\end{array}
$$

$$
\begin{array}{lll}
X & ::= & X_1, X_2, ...
\end{array}
$$

$$
\begin{array}{lll}
Traits & ::= & \varnothing \\
& | & \{Trait\} \cup Traits
\end{array}
$$

$$
\begin{array}{lll}
Trait & ::= & \{l : Type\} \qquad (\mathit{RecordLabel})
\end{array}
$$

Record Label A record label trait specifies a type T that a record must have associated to a label l. No bounds are placed on the size of the record, since records are unordered sets of label-type pairs.

To define the set of types that belong to a record label $\{l : T\}$, the following rules are used:

$$\{l_1 : T_1, \ \ldots \ l_n : T_n, \ \ldots \ T_k\} \in \{l : T\} \iff l_n = l \wedge T_n = T \qquad (1 \leq n \leq k)$$
$$X^{Traits} \in \{l : T\} \implies \{l : T\} \in Traits$$

2 Unnamed Section

Accessors are constructs that allow reading and writing to fields within a record.

In their most basic form, they consist of a single label (#*l*). In this state, an accessor creates a link to a specific field of a record, allowing it to be read from and written to directly.

One of the features that make accessors powerful is the fact that they can operate of different types of records.

In the following example, the same accessor (`#age`) is used on two different data types.

```
get #age {name: "John", age: 35}
// Getting the age of a person
get #age {type: "red", age: 12}
// Getting the age of wine
```

This is possible because accessors create a loose restriction on the record they access. Instead of requiring a specific type, they only require that the record conform to a trait. In the example above, the only requirement made by the `#age` accessor is that the record contain the field `age`.

Another feature of accessors is their ability to be "stacked". With this, it is easy to access arbitratily deep nested records.

In the example below, a stacked accessor is used to modify the street number of a record inside another record.

```
let address = {street: "Infinite Loop", number: 1}
let company = {name: "Apple Inc", headquarters: address}

set #headquarters.number 4 company
```

Stacked accessors, like their non-stacked brethren, also create the smallest restriction possible on its type. In the case above, the expression `#headquarters.number` specifies that the accessed record must have a field with the label `headquarters`.

It also specifies that this field is a record that, in turn, has a field with the label `number`. Furthermore, the fact that we are using the `set` function with an integer argument `4` tells us that the value associated with the label `number` is an integer.

Besides stacking, it is also possible to join accessors. Joined accessors operate on different fields of the same record. The values are treated as tuples, both for setting new values and for getting the current values.

Below is a simple example of accessing the fields `width` and `height` in a record.

```
get #(#width, #height) {width: 40, height: 30}
// Returns (40, 30)
set #(#width, #height) (60, 20) {width: 40, height: 30}
// Returns {width: 60, height: 20}
```

Joined accessors create multiple restrictions on the record they access. In the case above, the record is required to have both fields `width` and `height` (and, because of the `set` argument, we know these fields are integers).

When setting, joined accessors are "applied" from left to right. This means that, if multiple components of the accessor refer to the same field, the last component is used.

```
set #(#width, #width) (60, 20) {width: 40, height: 30}
// Returns {width: 20, height: 30}
```

It is possible to "distort" accessors, defining "getter" and "setter" functions to be applied on the values. These functions allow a value to be stored in a different format (or even type) than the one used when operating on it.

```
let ctoF x = x * 9 / 5 + 32;
let ftoC x = (x - 32) * 5 / 9;

let fahrenheit = #celsius [ctoF, ftoC];

get fahrenheit {celsius: 30}
// Returns 86
```

This distortion, combined with joined accessors, allows very powerful and expressive operations. In the example below, we define a custom accessor to return the area of a record.

```
let tupleToArea (x, y) = x + y;
let areaToTuple a = (a, 1);

let area = #(#width, #height) [tupleToArea, areaToTuple];

get area {width: 40, height: 30}
// Returns 1200
```

3 Evaluation

$$
\begin{array}{lll}
env & ::= & \{\} \mid \{x \rightarrow v\} \cup env \\
\\
v & ::= & n \\
 & \mid & raise \\
 & \mid & (v_1, \ \ldots \ v_n) \qquad (n \geq 2) \\
 & \mid & \{l_1 : v_1, \ \ldots \ l_n : v_n\} \qquad (n \geq 1) \\
 & \mid & \langle p, e, env \rangle \\
 & \mid & \#path \\
\\
path & ::= & l \\
 & \mid & path_1 \ . \ path_2 \\
 & \mid & (path_1, \ \ldots \ path_n) \qquad (n \geq 2) \\
 & \mid & path \ [v_1, \ v_2]
\end{array}
$$

3.1 Pattern Matching

$$match(x, v) = true, \{x \rightarrow v\}$$

$$match(\_, v) = true, \{\}$$

$$\frac{||n_1|| = ||n_2||}{match(n_1, n_2) = true, \{\}}$$

$$\frac{||n_1|| \neq ||n_2||}{match(n_1, n_2) = false, \{\}}$$

$$\frac{\exists \ i \in [1, n] \quad match(p_i, v_i) = false, env_i}{match((p_1, ...p_n), (v_1, ...v_n)) = false, \{\}}$$

$$\frac{\forall \ i \in [1, n] \quad match(p_i, v_i) = true, env_i}{match((p_1, ...p_n), (v_1, ...v_n)) = true, \bigcup_{i=1}^{n} env_i}$$

$$\frac{k \geq n \qquad \exists\ i \in [1,n] \quad \exists\ j \in [1,k] \quad l_i^1 = l_j^2 \wedge match(p_i, v_j) = false, env_i}{match(\{l_1^1 : p_1, \ldots, l_n^1 : p_n, \ldots\}, \{l_1^2 : v_1, \ldots l_k^2 : v_k\}) = false, \{\}}$$

$$\frac{k \geq n \qquad \forall\ i \in [1,n] \quad \exists\ j \in [1,k] \quad l_i^1 = l_j^2 \wedge match(p_i, v_j) = true, env_i}{match(\{l_1^1 : p_1, \ldots, l_n^1 : p_n, \ldots\}, \{l_1^2 : v_1, \ldots l_k^2 : v_k\}) = true, \bigcup_{i=1}^{n} env_i}$$

$$\frac{\exists\ i \in [1,n] \quad l_i^1 = l_i^2 \wedge match(p_i, v_i) = false, env_i}{match(\{l_1^1 : p_1, \ldots, l_n^1 : p_n\}, \{l_1^2 : v_1, \ldots l_n^2 : v_n\}) = false, \{\}}$$

$$\frac{\forall\ i \in [1,n] \quad l_i^1 = l_i^2 \wedge match(p_i, v_i) = true, env_i}{match(\{l_1^1 : p_1, \ldots, l_n^1 : p_n\}, \{l_1^2 : v_1, \ldots l_n^2 : v_n\}) = true, \bigcup_{i=1}^{n} env_i}$$

3.2 Path Evaluation Rules

$$\text{env} \vdash l \Downarrow l \qquad \text{(BS-Label)}$$

$$\frac{\text{env} \vdash path_1 \Downarrow path_1' \qquad \text{env} \vdash path_2 \Downarrow path_2'}{\text{env} \vdash path_1\ .\ path_2 \Downarrow path_1'\ .\ path_2'} \qquad \text{(BS-Stacked)}$$

$$\frac{\forall\ k \in [1,n] \quad \text{env} \vdash e_k \Downarrow \#path_k}{\text{env} \vdash (e_1,\ \ldots\ e_n) \Downarrow (path_1,\ \ldots\ path_n)} \qquad \text{(BS-Joined)}$$

$$\frac{\text{env} \vdash path_1 \Downarrow path' \qquad \text{env} \vdash e_1 \Downarrow v_1 \qquad \text{env} \vdash e_2 \Downarrow v_2}{\text{env} \vdash path\ [e_1,\ e_2] \Downarrow path\ [v_1,\ v_2]} \qquad \text{(BS-Distorted)}$$

3.3 Path Traversing Rules

$$\frac{1 \leq \|k\| \leq \|n\| \qquad r = \{l_1 : v_1,\ \ldots\ l_k : v,\ \ldots\ l_n : v_n\}}{traverse(l_k, \{l_1 : v_1, \ldots l_n : v_n\}, v) = v_k, r}$$

$$\frac{traverse(path_1, \{l_1 : v_1, \ldots l_n : v_n\}, v') = rec, r' \qquad traverse(path_2, rec, v) = v', rec'}{traverse(path_1\ .\ path_2, \{l_1 : v_1, \ldots l_n : v_n\}, v) = v', r'}$$

$$\frac{\{\} \vdash v_2\ v \Downarrow v' \qquad traverse(path, \{l_1 : v_1, \ldots l_n : v_n\}, v') = v'', r \qquad \{\} \vdash v_1\ v'' \Downarrow v'''}{traverse(path[v_1,\ v_2], \{l_1 : v_1, \ldots l_n : v_n\}, v) = v''', r}$$

$$\frac{\begin{array}{c} r_0 = \{l_1 : v_1, ...l_n : v_n\} \\ \forall i \in [1,\ n]\ .\ traverse(path_i, r_{i-1}, v_i) = v'_i, r_i \end{array}}{traverse((path_1,\ \ldots\ path_n), \{l_1 : v_1, ...l_n : v_n\}, (v_1,\ \ldots\ v_n)) = (v'_1,\ \ldots\ v'_n), r_n}$$

3.4 Big-Step Rules

$$\text{env} \vdash n \Downarrow n \qquad \text{(BS-Num)}$$

$$\text{env} \vdash \#path \Downarrow \#path \qquad \text{(BS-Access)}$$

$$\frac{\text{env}(x) = v}{\text{env} \vdash x \Downarrow v} \qquad \text{(BS-Ident)}$$

$$\frac{\forall\ k \in [1, n] \quad \text{env} \vdash e_k \Downarrow v_k}{\text{env} \vdash (e_1,\ \ldots\ e_n) \Downarrow (v_1,\ \ldots\ e_n)} \qquad \text{(BS-Tuple)}$$

Records A record construction expression $\{l_1 : e_1,\ \ldots\ l_n : e_n\}$ evaluates each of its sub-expressions individually, resulting in a record value.

$$\frac{\forall\ k \in [1, n] \quad \text{env} \vdash e_k \Downarrow v_k}{\text{env} \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} \Downarrow \{l_1 : v_1,\ \ldots\ l_n : v_n\}} \qquad \text{(BS-Record)}$$

$$\frac{\begin{array}{c} \text{env} \vdash e_1 \Downarrow \#path \qquad \text{env} \vdash e_2 \Downarrow \{l_1 : v_1,\ \ldots\ l_n : v_n\} \\ traverse(path, e_2, v) = v', r' \end{array}}{\text{env} \vdash get\ e_1\ e_2 \Downarrow v'} \qquad \text{(BS-Get)}$$

$$\frac{\begin{array}{c} \text{env} \vdash e_1 \Downarrow \#path \qquad \text{env} \vdash e_2 \Downarrow v \qquad \text{env} \vdash e_3 \Downarrow \{l_1 : v_1,\ \ldots\ l_n : v_n\} \\ traverse(path, e_2, v) = v', r' \end{array}}{\text{env} \vdash set\ e_1\ e_2\ e_3 \Downarrow r'} \qquad \text{(BS-Set)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \#path \qquad \text{env} \vdash v_1 \Downarrow v \qquad \text{env} \vdash e_3 \Downarrow v_2}{\text{env} \vdash distort\ e_1\ e_2\ e_3 \Downarrow \#path[v_1\ v_2]} \qquad \text{(BS-Distort)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \#path_1 \qquad \text{env} \vdash e_2 \Downarrow \#path_2}{\text{env} \vdash stack\ e_1\ e_2 \Downarrow \#path_1\ .\ path_2} \qquad \text{(BS-Stack)}$$

$$\text{env} \vdash \texttt{fn}\ p \Rightarrow e \Downarrow \langle p, e, env \rangle \qquad \text{(BS-Fn)}$$

Application An application expression requires either a closure or a recursive closure for its left-hand operand. The right-hand operand (argument) is always evaluated using the current environment, resulting in a value v_2.

In the case of a simple closure, the body of the function (e) is evaluated using the stored closure, matching the parameter pattern (p) with the argument (v_2).

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle p, e, env \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ match(p, v_2) = true, env_1 \\ env_1 \cup \text{env} \vdash e \Downarrow v\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow v} \qquad \text{(BS-AppFn)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle p, e, env \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ match(p, v_2) = false, env_1\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow raise} \qquad \text{(BS-AppFn2)}$$

Let Expressions These expressions are used to associate an identifier with a specific value, allowing the value to be reused throughout the program. Since V is a functional language, these are not variables, and the values assigned to an identifier will be constant (unless the same identifier is used in a new *let* expression).

After evaluating the expression that is to be associated to the identifier (that is, e_1), resulting in v, the *let* expression evaluates e_2. For this evaluation, the association of p to v is added to the environment. The result of this evaluation (that is, v_2) is the final result of the evaluation of the entire *let* expression.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow v \qquad match(p, v) = true, env_1 \\ env_1 \cup \text{env} \vdash e_2 \Downarrow v_2\end{array}}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow v_2} \qquad \text{(BS-Let)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad match(p, v) = false, env_1}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow raise} \qquad \text{(BS-Let2)}$$

Match Expression The match expression receives a input value and a list of *match*, attempting to pattern match against each one. The first *match* which returns a positive result is considered valid, and its corresponding expression is evaluted as the result of the whole expression.

If no *match* returns a positive result, the whole expression evaluates to *raise*.

$$\frac{\begin{array}{c}\text{env} \vdash e \Downarrow v \\ \exists j \in [1..n]\, multiMatch(v, \text{env}, match_j) = true, v_j \\ \forall k \in [1..j)\, multiMatch(v, \text{env}, match_k) = false, v_k\end{array}}{\text{env} \vdash \texttt{match}\ e\ \texttt{with}\ match_1, ...\ match_n \Downarrow v_j} \qquad \text{(BS-Match)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e \Downarrow v \\ \forall j \in [1..n]\, multiMatch(v, \text{env}, match_j) = false, v_j\end{array}}{\text{env} \vdash \texttt{match}\ e\ \texttt{with}\ match_1, ...\ match_n \Downarrow raise} \quad \text{(BS-Match2)}$$

In order to properly evaluate a match expression, it is necessary to define an auxiliary function, here called *multiMatch*. This function receives an input value, an environment and a *match*.

If the *match* has a conditional expression, it must evaluate to *true* for the match to be considered valid.

$$\frac{match(p, v) = false, env_1}{multiMatch(v, \text{env}, p \rightarrow e) = false, v}$$

$$\frac{match(p, v) = true, env_1 \qquad \text{env} \cup env_1 \vdash e \Downarrow v_2}{multiMatch(v, \text{env}, p \rightarrow e) = true, v_2}$$

$$\frac{match(p, v) = false, env_1}{multiMatch(v, \text{env}, p \text{ when } e_1 \rightarrow e_2) = false, v}$$

$$\frac{match(p, v) = true, env_1 \qquad \text{env} \cup env_1 \vdash e_1 \Downarrow false}{multiMatch(v, \text{env}, p \text{ when } e_1 \rightarrow e_2) = false, v}$$

$$\frac{\begin{array}{c}match(p, v) = true, env_1 \qquad \text{env} \cup env_1 \vdash e_1 \Downarrow true \\ \text{env} \cup env_1 \vdash e_2 \Downarrow v_2\end{array}}{multiMatch(v, \text{env}, p \text{ when } e_1 \rightarrow e_2) = true, v_2}$$

Exceptions Some programs can be syntactically correct but still violate the semantics of the *V* language, such as a dividing by zero or trying to access the head of an empty list. In these scenarios, the expression is evaluated as the *raise* value.

Besides violation of semantic rules, the only other expression that evaluates to the *raise* value is the *raise* expression, using the following rule:

$$\text{env} \vdash raise \Downarrow raise \quad \text{(BS-Raise)}$$

4 Type System

The type inference algorithm is divided into 3 parts:

1. Constraint Collection

   Collects constraints (defined below) for types in the program. This is a recursive operation, traversing the syntax tree and collecting constraints from the bottom up.

2. Constraint Unification

   Validates the collected constraints, resulting in a set of type and trait substitutions. If the constraints cannot be unified (i.e they are invalid), the type inference algorith ends here.

3. Unification Application

   Applies the substitutions the the type of the program. This guarantees that the result of the type inference is a base type (or variable types if the result is a polymorphic function).

4.1 Constraint Collection

Constraints are equations between type expressions, which can have both constant types and variable types. To infer the type of a program, the type system recursively collects a set of constraints for every subexpression in that program. This is done in a static way across the expression tree from the nodes to the root, without having to evaluate any of the expressions. To create a valid set of constraints, the system must contain an environment, built from the root to the nodes, to ensure identifiers are properly typed.

Environment Just like the operational semantics, the type system also uses an environment to store information about identifiers. In this case, the environment maps identifiers to type associations. These can be either simple associations or universal associations, which are used for polymorphic functions.

Simple Associations These associate an identifier with a unique type, which can be either constant or a variable type. When the association is called, the type is returned as-is, even if it is a variable type.

Universal Associations This association, also called a type scheme, stores a type which contains at least one variable type bound by a "for all" quantifier ($\forall$). When called, this association creates a new variable type for each bound variable and returns a new instance of the type scheme. Universal associations are used exclusively for polymorphic functions.

To create this type of association, the type system must generate a list of "free variables" present in the type that is to be universalized. These are the variable types that are not present in the environment when the identifier is declared. When these free variables are found, they are universally bound. This ensures that only those variable types that are unbound in the environment become universally bound in the resulting association.

Environment Application If an identifier is bound to a simple association, its value is returned directly from the environment.

If, on the other hand, an identifier is bound to a universal association, some work must be done for the new type. For every free variable in the association, a fresh variable type is declared and substituted in the resulting type.

4.1.1 Pattern Matching Rules

$$match(x, T) = \{\}, \{x \rightarrow T\}$$

$$match(n, T) = \{T = Int\}, \{\}$$

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

$$\frac{\forall i \in [1, n] \quad X_i \; is \; new \wedge match(p_i, X_i) = c_i, env_i}{match((p_1, ...p_n), T) = \{(X_i, ...X_n) = T\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} env_i}$$

$$\frac{\forall i \in [1, n] \quad X_i \; is \; new \wedge match(p_i, X_i) = c_i, env_i \qquad X_0^{\{\{l_i : X_i\}\} \forall i \in [1,n]}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}, T) = \{X_0 = T\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} env_i}$$

$$\frac{\forall i \in [1, n] \quad X_i \; is \; new \wedge match(p_i, X_i) = c_i, env_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\}, T) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} env_i}$$

4.1.2 Path Typing Rules

$$\frac{X_1 \; is \; new \qquad X_2^{\{\{l : X_1\}\}} \; is \; new}{traverse(\Gamma, l) = X_1, \; X_2, \; X_1 \mid \{\}}$$

$$\frac{\begin{array}{c} traverse(\Gamma, path_1) = IO_1, \; Rec_1, \; Store_1 \mid C_1 \\ traverse(\Gamma, path_2) = IO_2, \; Rec_2, \; Store_2 \mid C_2 \end{array}}{traverse(\Gamma, path_1 \; . \; path_2) = IO_2, \; Rec_1, \; Store_1 \mid C_1 \cup C_2 \cup \{Store_1 = IO_2\}}$$

$$\frac{\begin{array}{c} traverse(\Gamma, path) = IO, \; Rec, \; Store \mid C \\ \Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \\ X_{io} \; is \; new \qquad X_{store} \; is \; new \\ C' = \{T_1 = X_{store} \rightarrow X_{io}, T_2 = X_{io} \rightarrow X_{store}, X_{store} = IO\} \end{array}}{traverse(\Gamma, path \; [e_1, \; e_2]) = X_{io}, \; Rec, \; Store \mid C \cup C_1 \cup C_2 \cup C'}$$

$$\frac{\begin{array}{c} T = (X_1,\ \ldots\ X_n)\ \textit{is new} \qquad X_{rec}\ \textit{is new} \\ \forall i \in [1,n]\ .\ \Gamma \vdash e_i : T_i \mid C_i \\ \forall i \in [1,n]\ .C'_i = C \cup \{T_i = \#(X_i \to X_{rec})\} \end{array}}{\textit{traverse}(\Gamma, (e_1,\ \ldots\ e_n)) = T,\ X_{rec},\ T \mid \bigcup_{i=1}^{n} C'_i}$$

4.1.3 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.

$$\Gamma \vdash n : \text{Int} \mid \{\} \qquad \text{(T-Num)}$$

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

$$\frac{\forall\ k \in [1,n] \quad \Gamma \vdash e_k : T_k \mid C_k}{\Gamma \vdash (e_1,\ \ldots\ e_n) : (T_1,\ \ldots\ T_n) \mid C_1 \cup \cdots C_n} \qquad \text{(T-Tuple)}$$

$$\frac{\forall\ k \in [1,n] \quad \Gamma \vdash e_k : T_k \mid C_k}{\Gamma \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} : \{l_1 : T_1,\ \ldots\ l_n : T_n\} \mid C_1 \cup \cdots C_n} \qquad \text{(T-Record)}$$

$$\frac{X_1\ \textit{is new} \qquad \textit{traverse}(\textit{paths}, X_1) = IO, Rec, , C}{\Gamma \vdash \#\textit{paths} : \#(IO \to Rec) \mid \{\}} \qquad \text{(T-Access)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1\ \textit{is new}}{\Gamma \vdash \textit{get}\ e_1\ e_2 : X_1 \mid C_1 \cup C_2 \cup \{T_1 = \#(X_1 \to T_2)\}} \qquad \text{(T-Get)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad \Gamma \vdash e_3 : T_3 \mid C_3}{\Gamma \vdash \textit{set}\ e_1\ e_2\ e_3 : T_3 \mid C_1 \cup C_2 \cup C_3 \cup \{T_1 = \#(T_2 \to T_3)\}} \qquad \text{(T-Set)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \\ X_1\ \textit{is new} \qquad X_2\ \textit{is new} \qquad X_3\ \textit{is new} \end{array}}{\Gamma \vdash \textit{stack}\ e_1\ e_2 : \#(X_3 \to X_2) \mid C_1 \cup C_2 \cup \{T_1 = \#(X_1 \to X_2), T_2 = \#(X_3 \to X_1)\}} \qquad \text{(T-Stack)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad \Gamma \vdash e_3 : T_3 \mid C_3 \\ X_1\ \textit{is new} \qquad X_2\ \textit{is new} \qquad X_3\ \textit{is new} \end{array}}{\Gamma \vdash \textit{distort}\ e_1\ e_2\ e_3 : \#(X_3 \to X_2) \mid C_1 \cup C_2 \cup C_3 \cup \{T_1 = \#(X_1 \to X_2), T_2 = X_1 \to X_3, T_3 = X_3 \to X_1\}} \qquad \text{(T-Distort)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1\ \textit{is new}}{\Gamma \vdash e_1\ e_2 : X \mid C_1 \cup C_2 \cup \{T1 = T_2 \to X_1\}} \qquad \text{(T-App)}$$

$$\frac{X \text{ is new} \qquad match(p, X) = C, env \qquad env \cup \Gamma \vdash e : T_1 \mid C_1}{\Gamma \vdash \texttt{fn}\ p \Rightarrow e : X \to T_1 \mid C \cup C_1} \quad \text{(T-Fn)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T \mid C \qquad X_1 \text{ is new} \\ \forall j \in [1..n]\, multiMatch(T, X_1, \Gamma, match_j) = C_j\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ match_1, ... \ match_n : X_1 \mid C \cup \bigcup_{i=1}^{n} C_i} \quad \text{(T-Match)}$$

$$\frac{match(p, T_1) = C, \Gamma_1 \qquad \Gamma_1 \cup \Gamma \vdash e : T_3 \mid C_3}{multiMatch(T_1, T_2, \Gamma, p \to e) = C \cup C_3 \cup \{T_3 = T_2\}}$$

$$\frac{match(p, T_1) = C, \Gamma_1 \qquad \Gamma_1 \cup \Gamma \vdash e_1 : T_3 \mid C_3 \qquad \Gamma_1 \cup \Gamma \vdash e_2 : T_4 \mid C_4}{multiMatch(T_1, T_2, \Gamma, p \text{ when } e_1 \to e_2) = C \cup C_3 \cup C_4 \cup \{T_3 = Bool, T_4 = T_2}$$

$$\frac{X_1 \text{ is new}}{\Gamma \vdash raise : X_1 \mid \{\}} \quad \text{(T-Raise)}$$

The constraint collection rule for `let` is a little more complicated, as it envolves calling the type inference algorithm recursively. The steps to collect constraints for a expression $\texttt{let}\ p = e_1\ \texttt{in}\ e_2$ are as follow:

1. $\Gamma \vdash e_1 : T_1 \mid C_1$
2. $\texttt{unify}(C_1) = \sigma_1, \theta_1$
3. $\texttt{apply}(\sigma_1, \theta_1, T_1) = T_1'$
4. $\texttt{apply}(\sigma_1, \theta_1, \Gamma) = \Gamma'$
5. $\texttt{freeVars}(T_1', \Gamma') = free$
6. $\texttt{if} free = \varnothing \texttt{then}$
   (a) $match(p, T_1) = C, env$
7. `else`
   (a) $C = \varnothing$
   (b) $env = \Gamma \cup \{x \to \forall X \in free\ .\ T_1'\}$
8. $env \cup \Gamma \vdash e_2 : T_2 \mid C_2$
9. return $T_2 \mid C \cup C_1 \cup C_2$

4.2 Constraint Unification

The unification algorithm recursively iterates through the list of constraints, matching

$$\texttt{unify}(\varnothing) = \varnothing, \varnothing$$

$$\frac{T_1 = T_2}{\texttt{unify}(\{T_1 = T_2\} \cup C) = \texttt{unify}(C)}$$

$$\frac{T_1 = T_1^1 \to T_2^1 \qquad T_2 = T_1^2 \to T_2^2}{\texttt{unify}(\{T_1 = T_2\} \cup C) = \texttt{unify}(C \cup \{T_1^1 = T_1^2, T_2^1 = T_2^2\})}$$

$$\frac{T_1 = (T_1^1,\ \ldots\ T_n^1) \qquad T_2 = (T_1^2,\ \ldots\ T_n^2)}{\texttt{unify}(\{T_1 = T_2\} \cup C) = \texttt{unify}(C \cup \{T_1^1 = T_1^2,\ \ldots\ T_n^1 = T_n^2\})}$$

$$\frac{T_1 = \{l_1 : T_1^1,\ \ldots\ l_n : T_n^1\} \qquad T_2 = \{l_1 : T_1^2,\ \ldots\ l_n : T_n^2\}}{\texttt{unify}(\{T_1 = T_2\} \cup C) = \texttt{unify}(C \cup \{T_1^1 = T_1^2,\ \ldots\ T_n^1 = T_n^2\})}$$

$$\frac{\begin{array}{cc} T_1 = X^{tr} & X \notin \texttt{freeVars}(T_2, \varnothing) \\ T_2 \in tr & C' = [X \to T_2]C \end{array}}{\texttt{unify}(\{T_1 = T_2\} \cup C) = \texttt{unify}(C')}$$

$$\frac{\begin{array}{cc} T_2 = X^{tr} & X \notin \texttt{freeVars}(T_1, \varnothing) \\ T_1 \in tr & C' = [X \to T_1]C \end{array}}{\texttt{unify}(\{T_1 = T_2\} \cup C') = \texttt{unify}(C' \cup \{T_1' = T_2', T_1'' = T_2''\})}$$