1 Formalization

1.1 Syntax

$$f \in \textsc{TransformerNames} \qquad t \in \textsc{TypeNames}$$
$$a, x, y, z \in \textsc{Identifiers}$$

$$
\begin{array}{lll}
\mathcal{Q}, \mathcal{R}, \mathcal{S} & ::= & \texttt{!} \mid \texttt{any} \mid \texttt{nonempty} \mid \texttt{empty} \mid \texttt{every} \\
M & ::= & \texttt{fungible} \mid \texttt{unique} \mid \texttt{immutable} \mid \texttt{consumable} \mid \texttt{asset} \quad \text{(type declaration modifiers)} \\
T & ::= & \texttt{bool} \mid \texttt{nat} \mid \texttt{type}\ t\ \texttt{is}\ \overline{M}\ T \mid \texttt{list}\ \tau \mid \{\overline{x : \tau}\} \\
\tau, \sigma, \pi & ::= & \mathcal{Q}\ T \\
\mathcal{S} & ::= & x \mid x.y \mid \texttt{true} \mid \texttt{false} \mid [x] \mid \{\overline{x : \tau \mapsto x}\} \mid \texttt{new}(t, \overline{M}, T) \\
\mathcal{D} & ::= & x \mid x.y \mid \texttt{var}\ x : T \mid \texttt{consume} \\
\texttt{Decl} & ::= & \texttt{transformer}\ f(\overline{x : \tau}) \rightarrow x : \tau\ \{\overline{\texttt{Stmt}}\} \\
\texttt{Stmt} & ::= & \texttt{pass} \\
 & \mid & \mathcal{S} \rightarrow \mathcal{D} \mid \mathcal{S} \xrightarrow{\mathcal{Q}\ \textbf{s.t.}\ f(\overline{x})} \mathcal{D} \mid \mathcal{S} \rightarrow f(\overline{x}) \rightarrow \mathcal{D} \\
 & \mid & \texttt{try}\ \{\overline{\texttt{Stmt}}\}\ \texttt{catch}\ \{\overline{\texttt{Stmt}}\} \\
\texttt{Prog} & ::= & \overline{\texttt{Decl}}; \overline{\texttt{Stmt}}
\end{array}
$$

[Include flow-by-variable?]

1.2 Statics

$\boxed{\Gamma \vdash \mathcal{S} : \tau \dashv \Delta}$ $\boxed{\Gamma \vdash \mathcal{D} : \tau \dashv \Delta}$ **Storage Typing**

A *storage* is either a source or a destination.

$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash b : \texttt{!}\ \texttt{bool} \dashv \Gamma}\ \textsc{Bool} \qquad \frac{}{\Gamma \vdash n : \texttt{!}\ \texttt{nat} \dashv \Gamma}\ \textsc{Nat} \qquad \frac{\neg(\tau\ \texttt{immutable})}{\Gamma, x : \tau \vdash x : \tau \dashv \Gamma, x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma \vdash x : \tau \dashv \Delta \qquad \neg(\tau\ \texttt{immutable}) \qquad \texttt{fields}(\tau) = \overline{z : \sigma} \qquad y : \mathcal{R}\ T \in \overline{z : \sigma}}{\Gamma \vdash x.y : \mathcal{R}\ T \dashv \Gamma}\ \textsc{Field}$$

$$\frac{}{\Gamma, x : \mathcal{Q}\ T \vdash [x] : \texttt{!}\ \texttt{list}\ \mathcal{Q}\ T \dashv \Gamma, x : \texttt{empty}\ T}\ \textsc{Single}$$

$$\frac{}{\Gamma, \overline{y : \mathcal{Q}\ T} \vdash \{\overline{x : \mathcal{Q}\ T \mapsto y}\} : \texttt{!}\ \{\overline{x : \mathcal{Q}\ T}\} \dashv \Gamma, \overline{y : \texttt{empty}\ T}}\ \textsc{Record}$$

$$\frac{}{\Gamma \vdash \texttt{new}(t, \overline{M}, T) : \texttt{every}\ \texttt{list}\ \texttt{!}\ (\texttt{type}\ t\ \texttt{is}\ \overline{M}\ T) \dashv \Gamma}\ \textsc{New}$$

$$\frac{}{\Gamma \vdash (\texttt{var}\ x : T) : \texttt{empty}\ T \dashv \Gamma, x : \texttt{empty}\ T}\ \textsc{VarDef} \qquad \frac{\tau\ \texttt{consumable}}{\Gamma \vdash \texttt{consume} : \tau \dashv \Gamma}\ \textsc{Consume}$$

$\boxed{\Gamma \vdash S \ \textbf{ok} \dashv \Delta}$ **Statement Well-formedness**

$$\frac{}{\Gamma \vdash \texttt{pass}\ \textbf{ok} \dashv \Gamma}\ \text{Ok-Pass} \qquad \frac{\Gamma \vdash \mathcal{S} : \mathcal{Q}\ T \dashv \Delta \qquad \texttt{update}(\Delta, \mathcal{S}, \Delta(\mathcal{S}) \ominus \mathcal{Q}) \vdash \mathcal{D} : \mathcal{R}\ T \dashv \Xi}{\Gamma \vdash (\mathcal{S} \to \mathcal{D})\ \textbf{ok} \dashv \texttt{update}(\Xi, \mathcal{D}, \Xi(\mathcal{D}) \oplus \mathcal{Q})}\ \text{Ok-Flow-Every}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{S} : \mathcal{Q}\ T \dashv \Delta \\ \texttt{transformer}\ f(\overline{x : \sigma}, y : \text{demote}(\texttt{elemtype}(T))) \to z : !\ \texttt{bool}\ \textbf{do}\ \overline{\texttt{Stmt}} \\ \forall i.\text{demote}(\Gamma(a_i)) = \sigma_i \qquad \texttt{update}(\Delta, \mathcal{S}, \Delta(\mathcal{S}) \ominus \mathcal{Q}) \vdash \mathcal{D} : \mathcal{S}\ T \dashv \Xi\end{array}}{\Gamma \vdash (\mathcal{S} \xrightarrow{\mathcal{R}\ \textbf{s.t.}\ f(\overline{a})} \mathcal{D})\ \textbf{ok} \dashv \texttt{update}(\Xi, \mathcal{D}, \Xi(\mathcal{D}) \oplus \min(\mathcal{Q}, \mathcal{R}))}\ \text{Ok-Flow-Filter}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{S} : \mathcal{Q}\ T_1 \dashv \Delta \\ \texttt{transformer}\ f(\overline{x : \sigma}, y : \text{demote}(\texttt{elemtype}(T_1))) \to z : \mathcal{R}\ T_2\ \textbf{do}\ \overline{\texttt{Stmt}} \\ \forall i.\text{demote}(\Gamma(x_i)) = \sigma_i \qquad \texttt{update}(\Delta, \mathcal{S}, \Delta(\mathcal{S}) \ominus \mathcal{Q}) \vdash \mathcal{D} : \mathcal{S}\ T_2 \dashv \Xi\end{array}}{\Gamma \vdash (\mathcal{S} \to f(\overline{x}) \to \mathcal{D})\ \textbf{ok} \dashv \texttt{update}(\Xi, \mathcal{D}, \Xi(\mathcal{D}) \oplus \mathcal{Q})}\ \text{Ok-Flow-Transformer}$$

$$\frac{\Gamma \vdash \overline{S_1}\ \textbf{ok} \dashv \Delta \qquad \Gamma \vdash \overline{S_2}\ \textbf{ok} \dashv \Xi}{\Gamma \vdash (\texttt{try}\ \{\overline{S_1}\}\ \texttt{catch}\ \{\overline{S_2}\}\ \textbf{ok} \dashv \Delta \sqcup \Xi}\ \text{Ok-Try}$$

$\boxed{\vdash \texttt{Decl}\ \textbf{ok}}$ **Declaration Well-formedness**

$$\frac{\overline{x : \tau} \vdash \overline{\texttt{Stmt}}\ \textbf{ok} \dashv \Gamma, y : \sigma \qquad \forall \pi \in \texttt{img}(\Gamma).\neg(\pi\ \texttt{asset})}{\vdash (\texttt{transformer}\ f(\overline{x : \tau}) \to y : \sigma\{\overline{\texttt{Stmt}}\})\ \textbf{ok}}\ \text{Ok-Transformer}$$

$\boxed{\texttt{Prog}\ \textbf{ok}}$ **Program Well-formedness**

$$\frac{\vdash \overline{\texttt{Decl}}\ \textbf{ok} \qquad \emptyset \vdash \overline{\texttt{Stmt}}\ \textbf{ok} \dashv \Gamma \qquad \forall \tau \in \texttt{img}(\Gamma).\neg(\tau\ \texttt{asset})}{(\overline{\texttt{Decl}}; \overline{\texttt{Stmt}})\ \textbf{ok}}\ \text{Ok-Prog}$$

1.3 Dynamics

$$\begin{array}{lcl} V & ::= & \texttt{true} \mid \texttt{false} \mid n \mid \{x : \tau \mapsto V\} \\ \mathcal{V} & ::= & \overline{V} \\ \texttt{Stmt} & ::= & \ldots \mid \texttt{put}(\mathcal{V}, \mathcal{D}) \mid \texttt{revert} \mid \texttt{try}(\Sigma, \overline{S}, \overline{S}) \end{array}$$

Definition 1. *An environment* Σ *is a tuple* (μ, ρ) *where* μ : $\textit{IdentifierNames} \rightharpoonup \mathbb{N}$ *is the* variable lookup environment, *and* $\rho : \mathbb{N} \rightharpoonup \mathcal{V}$ *is the* storage environment.

$\boxed{\left\langle \Sigma, \overline{\texttt{Stmt}} \right\rangle \to \left\langle \Sigma, \overline{\texttt{Stmt}} \right\rangle}$

Note that we abbreviate $\langle \Sigma, \cdot \rangle$ as Σ, which signals the end of evaluation.

The new constructs of $\texttt{resolve}(\Sigma, \mathcal{S})$ and $\texttt{put}(\mathcal{V}, \mathcal{D})$ are used to simplify the process of locating sources and updating destinations, respectively.

$$\frac{\langle \Sigma, S_1 \rangle \to \left\langle \Sigma', \overline{S_3} \right\rangle}{\left\langle \Sigma, S_1 \overline{S_2} \right\rangle \to \left\langle \Sigma', \overline{S_3}\ \overline{S_2} \right\rangle}\ \text{Seq} \qquad \frac{}{\left\langle \Sigma, (\texttt{revert})\ \overline{S} \right\rangle \to \langle \Sigma, \texttt{revert} \rangle}\ \text{Revert} \qquad \frac{}{\langle \Sigma, \texttt{pass} \rangle \to \Sigma}\ \text{Pass}$$

Here we give the rules for the new $\texttt{put}(\mathcal{V}, \mathcal{D})$ statement. **[TODO: Need to finalize how $\mathcal{V} + \mathcal{W}$ works; in particular, need to make sure that you can't overwrite things that shouldn't be overwritten (e.g., a nonfungible nat). Probably need to tag types with modifiers or something.]**

$$\frac{}{\langle \Sigma, \mathtt{put}(\mathcal{V}, \mathsf{consume})\rangle \to \Sigma}\ \textsc{Put-Consume} \qquad \frac{\rho(\mu(A)) = \mathcal{W} \qquad \mathcal{W} + \mathcal{V} \neq \mathtt{revert}}{\langle \Sigma, \mathtt{put}(\mathcal{V}, A)\rangle \to \Sigma[\rho \mapsto \rho[\mu(A) \mapsto \mathcal{W} + \mathcal{V}]]}\ \textsc{Put-Var}$$

$$\frac{\rho(\mu(A)) = \mathcal{W} \qquad \mathcal{W} + \mathcal{V} = \mathtt{revert}}{\langle \Sigma, \mathtt{put}(\mathcal{V}, A)\rangle \to \langle \Sigma, \mathtt{revert}\rangle}\ \textsc{Put-Var-Fail}$$

$$\frac{\ell \notin \mathsf{dom}(\rho)}{\langle \Sigma, \mathtt{put}(\mathcal{V}, \mathtt{var}\ A : T)\rangle \to \Sigma[\mu \mapsto \mu[A \mapsto \ell], \rho \mapsto \rho[\ell \mapsto \mathcal{V}]]}\ \textsc{Put-VarDef}$$

$$\frac{\mathtt{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell)}{\langle \Sigma, \mathcal{S} \to \mathcal{D}\rangle \to \langle \Sigma'[\rho \mapsto \rho'[\ell \mapsto []]], \mathtt{put}(\rho'(\ell), \mathcal{D})\rangle}\ \textsc{Flow-Every}$$

$$\frac{\begin{array}{c}\mathtt{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \qquad \rho'(\ell) = \mathcal{V} \\ \mathcal{U} = [v \in \mathcal{V} \mid \langle \Sigma', f(\overline{x}, v)\rangle \to^* \langle \Sigma'', k\rangle \text{ and } \rho''(k) = \mathtt{true}] \qquad \mathtt{compat}(|\mathcal{U}|, |\mathcal{V}|, \mathcal{Q})\end{array}}{\Sigma, (\mathcal{S} \xrightarrow{\mathcal{Q}\ \textbf{s.t.}\ f(\overline{x})} \mathcal{D}) \to \langle \Sigma'[\rho' \mapsto \rho'[\ell \mapsto \rho'(\ell) - \mathcal{U}]], \mathtt{put}(\mathcal{U}, \mathcal{D})\rangle}\ \textsc{Flow-Filter}$$

$$\frac{\begin{array}{c}\mathtt{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \qquad \rho'(\ell) = \mathcal{V} \\ \mathcal{U} = [v \in \mathcal{V} \mid \langle \Sigma', f(\overline{x}, v)\rangle \to^* \langle \Sigma'', k\rangle \text{ and } \rho''(k) = \mathtt{true}] \qquad \neg\mathtt{compat}(|\mathcal{U}|, |\mathcal{V}|, \mathcal{Q})\end{array}}{\Sigma, (\mathcal{S} \xrightarrow{\mathcal{Q}\ \textbf{s.t.}\ f(\overline{x})} \mathcal{D}) \to \langle \Sigma', \mathtt{revert}\rangle}\ \textsc{Flow-Filter-Fail}$$

$$\frac{\begin{array}{c}\mathtt{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \\ \rho'(\ell) = v, \mathcal{V} \qquad \langle \Sigma'[\rho \mapsto \rho'[\ell \mapsto \mathcal{V}]], f(\overline{x}, v)\rangle \to^* \langle \Sigma'', k\rangle\end{array}}{\langle \Sigma, \mathcal{S} \to f(\overline{x}) \to \mathcal{D}\rangle \to \langle (\mu', \rho''), \mathtt{put}(\rho''(k), \mathcal{D})\ (\mathcal{S} \to f(\overline{x}) \to \mathcal{D})\rangle}\ \textsc{Flow-Transformer}$$

$$\frac{\mathtt{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \qquad \rho'(\ell) = []}{\langle \Sigma, \mathcal{S} \to f(\overline{x}) \to \mathcal{D}\rangle \to \Sigma}\ \textsc{Flow-Transformer-Done}$$

$$\frac{\ell \notin \mathsf{dom}(\rho) \qquad \mathtt{transformer}\ f(\overline{y : \tau}) \to z : \sigma\ \mathtt{do}\ \overline{S} \qquad \mu' = \overline{y \mapsto \mu(x)}, z \mapsto \ell}{\langle \Sigma, f(\overline{x})\rangle \to \left\langle (\mu', \rho[\ell \mapsto []]), \overline{S}\ \ell\right\rangle}\ \textsc{Call}$$

We introduce a new statement, $\mathtt{try}(\Sigma, \overline{S_1}, \overline{S_2})$, to implement the try-catch statement, which keeps track of the environment that we begin execution in so that we can revert to the original environment in the case of a $\mathtt{revert}$.

$$\frac{}{\left\langle \Sigma, \mathtt{try}\ \{\overline{S_1}\}\ \mathtt{catch}\ \{\overline{S_2}\}\right\rangle \to \left\langle \Sigma, \mathtt{try}(\Sigma, \overline{S_1}, \overline{S_2})\right\rangle}\ \textsc{Try-Start}$$

$$\frac{\left\langle \Sigma, \overline{S_1}\right\rangle \to \left\langle \Sigma'', \overline{S_1'}\right\rangle}{\left\langle \Sigma, \mathtt{try}(\Sigma', \overline{S_1}, \overline{S_2})\right\rangle \to \left\langle \Sigma'', \mathtt{try}(\Sigma', \overline{S_1'}, \overline{S_2})\right\rangle}\ \textsc{Try-Step}$$

$$\frac{}{\left\langle \Sigma, \mathtt{try}(\Sigma', \mathtt{revert}, \overline{S_2})\right\rangle \to \left\langle \Sigma', \overline{S_2}\right\rangle}\ \textsc{Try-Revert} \qquad \frac{}{\left\langle \Sigma, \mathtt{try}(\Sigma', \cdot, \overline{S_2})\right\rangle \to \Sigma}\ \textsc{Try-Done}$$

[Need to handle fungible specially (or maybe only after adding nats, I'm not sure it really has any meaning without them)]

$\boxed{\texttt{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell)}$ **Storage Resolution**

We use $\texttt{resolve}(\Sigma, \mathcal{S})$ to get the location storing the values of $\mathcal{S}$, which returns an environment because it may need to allocate new memory (e.g., in the case of creating a new record value).

$$\frac{\mu(\mathcal{S}) = \ell}{\texttt{resolve}(\Sigma, \mathcal{S}) = (\Sigma, \ell)}\ \textsc{Resolve-Var} \qquad \frac{\rho(\mu(x)) = \{\overline{z : \tau \mapsto \ell}\} \qquad (y : \sigma \mapsto k) \in \overline{z : \tau \mapsto \ell}}{\texttt{resolve}(\Sigma, x.y) = (\Sigma, k)}\ \textsc{Resolve-Field}$$

$$\frac{\ell \notin \mathsf{dom}(\rho)}{\texttt{resolve}(\Sigma, [x]) = (\Sigma[\rho \mapsto \rho[\ell \mapsto \rho(\mu(x)), \mu(x) \mapsto []]], \ell)}\ \textsc{Resolve-Single}$$

$$\frac{k \notin \mathsf{dom}(\rho) \cup \overline{\ell} \qquad \begin{array}{c}\overline{\ell \notin \mathsf{dom}(\rho)} \\ \Sigma' = \Sigma[\rho \mapsto \rho[\overline{\mu(y) \mapsto []}, \overline{\ell \mapsto \rho(\mu(y))}, k \mapsto \{\overline{x : \tau \mapsto \ell}\}]]\end{array}}{\texttt{resolve}(\Sigma, \{\overline{x : \tau \mapsto y}\}) = (\Sigma', k)}\ \textsc{Resolve-Record}$$

$$\frac{b \in \{\texttt{true}, \texttt{false}\} \qquad \ell \notin \mathsf{dom}(\rho)}{\texttt{resolve}(\Sigma, b) = (\Sigma[\rho \mapsto \rho[\ell \mapsto b]], \ell)}\ \textsc{Resolve-Bool}$$

$$\frac{\mu(t) = \ell}{\texttt{resolve}(\Sigma, \mathsf{new}(t, \overline{M}, T)) = (\Sigma, \ell)}\ \textsc{Resolve-Source}$$

$$\frac{t \notin \mathsf{dom}(\mu) \qquad \ell \notin \mathsf{dom}(\rho)}{\texttt{resolve}(\Sigma, \mathsf{new}(t, \overline{M}, T)) = (\Sigma'[\rho \mapsto \rho[\ell \mapsto \texttt{values}(T)], \mu \mapsto \mu[t \mapsto \ell]], \ell)}\ \textsc{Resolve-New-Source}$$

[TODO: Need to be sure to handle uniqueness correctly; could do this in Resolve-New-Source, or in the various flow rules.]

1.4 Auxiliaries

Definition 2. *Define* ***Quant*** $= \{\texttt{empty}, \texttt{any}, !, \texttt{nonempty}, \texttt{every}\}$*, and call any* $\mathcal{Q} \in$ ***Quant*** *a* type quantity. *Define* $\texttt{empty} < \texttt{any} < ! < \texttt{nonempty} < \texttt{every}$.

$\boxed{\tau\ \texttt{asset}}$ **Asset Types**

$$\begin{aligned}(\mathcal{Q}\ T)\ \texttt{asset} \Leftrightarrow\ & \mathcal{Q} \neq \texttt{empty} \text{ and } (\texttt{asset} \in \texttt{modifiers}(T) \text{ or} \\ & (T = \mathcal{C}\ \tau \text{ and } \tau\ \texttt{asset}) \text{ or} \\ & (T = \{\overline{y : \sigma}\} \text{ and } \exists x : \tau \in \overline{y : \sigma}.(\tau\ \texttt{asset})))\end{aligned}$$

$\boxed{\tau\ \texttt{consumable}}$ **Consumable Types**

$$\begin{aligned}(\mathcal{Q}\ T)\ \texttt{consumable} \Leftrightarrow\ & \texttt{consumable} \in \texttt{modifiers}(T) \text{ or} \\ & \neg((\mathcal{Q}\ T)\ \texttt{asset}) \text{ or} \\ & (T = \mathcal{C}\ \tau \text{ and } \tau\ \texttt{consumable}) \text{ or} \\ & (T = \{\overline{y : \sigma}\} \text{ and } \forall x : \tau \in \overline{y : \sigma}.(\sigma\ \texttt{consumable}))\end{aligned}$$

$\mathcal{Q} \oplus \mathcal{R}$ represents the quantity present when flowing $\mathcal{R}$ of something to a storage already containing $\mathcal{Q}$. $\mathcal{Q} \ominus \mathcal{R}$ represents the quantity left over after flowing $\mathcal{R}$ from a storage containing $\mathcal{Q}$.

Definition 3. *Let $\mathcal{Q},\mathcal{R} \in$ **Quant**. Define the commutative operator $\oplus$, called* combine, *as the unique function* $\boldsymbol{Quant}^2 \to \boldsymbol{Quant}$ *such that*

$$
\begin{aligned}
\mathcal{Q} \oplus \mathbf{empty} &= \mathcal{Q} \\
\mathcal{Q} \oplus \mathbf{every} &= \mathbf{every} \\
\mathbf{nonempty} \oplus \mathcal{R} &= \mathbf{nonempty} \quad \text{if } \mathbf{empty} < \mathcal{R} < \mathbf{every} \\
! \oplus \mathcal{R} &= \mathbf{nonempty} \quad \text{if } \mathbf{empty} < \mathcal{R} < \mathbf{every} \\
\mathbf{any} \oplus \mathbf{any} &= \mathbf{any}
\end{aligned}
$$

Define the operator $\ominus$, called split, *as the unique function* $\boldsymbol{Quant}^2 \to \boldsymbol{Quant}$ *such that*

$$
\begin{aligned}
\mathcal{Q} \ominus \mathbf{empty} &= \mathcal{Q} \\
\mathbf{empty} \ominus \mathcal{R} &= \mathbf{empty} \\
\mathcal{Q} \ominus \mathbf{every} &= \mathbf{empty} \\
\mathbf{every} \ominus \mathcal{R} &= \mathbf{every} \quad \text{if } \mathcal{R} < \mathbf{every} \\
\mathbf{nonempty} - \mathcal{R} &= \mathbf{any} \quad \text{if } \mathbf{empty} < \mathcal{R} < \mathbf{every} \\
! - \mathcal{R} &= \mathbf{empty} \quad \text{if } ! \leq \mathcal{R} \\
! - \mathbf{any} &= \mathbf{any} \\
\mathbf{any} - \mathcal{R} &= \mathbf{any} \quad \text{if } \mathbf{empty} < \mathcal{R} < \mathbf{every}
\end{aligned}
$$

Note that we write $(\mathcal{Q}\ T) \oplus \mathcal{R}$ to mean $(\mathcal{Q} \oplus \mathcal{R})\ T$ and similarly $(\mathcal{Q}\ T) \ominus \mathcal{R}$ to mean $(\mathcal{Q} \ominus \mathcal{R})\ T$.

Definition 4. *We can consider a type environment Γ as a function* $\textsc{Identifiers} \to \textsc{Types} \cup \{\bot\}$ *as follows:*

$$
\Gamma(x) = \begin{cases} \tau & \text{if } x : \tau \in \Gamma \\ \bot & \text{otherwise} \end{cases}
$$

We write $\mathbf{dom}(\Gamma)$ *to mean* $\{x \in \textsc{Identifiers} \mid \Gamma(x) \neq \bot\}$, *and* $\Gamma|_X$ *to mean the environment* $\{x : \tau \in \Gamma \mid x \in X\}$ *(restricting the domain of Γ).*

Definition 5. *Let $\mathcal{Q}$ and $\mathcal{R}$ be type quantities, $T_{\mathcal{Q}}$ and $T_{\mathcal{R}}$ base types, and Γ and Δ type environments. Define the following orderings, which make types and type environments into join-semilattices. For type quantities, define the partial order $\sqsubseteq$ as the reflexive closure of the strict partial order $\sqsubset$ given by*

$$
\mathcal{Q} \sqsubset \mathcal{R} \Leftrightarrow (\mathcal{Q} \neq \mathbf{any} \text{ and } \mathcal{R} = \mathbf{any}) \text{ or } (\mathcal{Q} \in \{!, \mathbf{every}\} \text{ and } \mathcal{R} = \mathbf{nonempty})
$$

For types, define the partial order $\leq$ by

$$
\mathcal{Q}\ T_{\mathcal{Q}} \leq \mathcal{R}\ T_{\mathcal{R}} \Leftrightarrow T_{\mathcal{Q}} = T_{\mathcal{R}} \text{ and } \mathcal{Q} \sqsubseteq \mathcal{R}
$$

For type environments, define the partial order $\leq$ by

$$
\Gamma \leq \Delta \Leftrightarrow \forall x. \Gamma(x) \leq \Delta(x)
$$

Denote the join of Γ and Δ by $\Gamma \sqcup \Delta$.

$\boxed{\texttt{elemtype}(T) = \tau}$

$$
\texttt{elemtype}(T) = \begin{cases} \texttt{elemtype}(T') & \text{if } T = \texttt{type}\ t\ \texttt{is}\ \overline{M}\ T' \\ \tau & \text{if } T = \mathcal{C}\ \tau \\ !\ T & \text{otherwise} \end{cases}
$$

$\boxed{\texttt{modifiers}(T) = \overline{M}}$ **Type Modifiers**

$$\texttt{modifiers}(T) = \begin{cases} \overline{M} & \text{if } T = \texttt{type } t \texttt{ is } \overline{M}\ T \\ \emptyset & \text{otherwise} \end{cases}$$

$\boxed{\text{demote}(\tau) = \sigma}$ $\boxed{\text{demote}_*(T_1) = T_2}$ **Type Demotion** demote and demote$_*$ take a type and "strip" all the asset modifiers from it, as well as unfolding named type definitions. This process is useful, because it allows selecting asset types without actually having a value of the desired asset type. Note that demoting a transformer type changes nothing. This is because a transformer is **never** an asset, regardless of the types that it operators on, because it has no storage.

$$\begin{aligned} \text{demote}(\mathcal{Q}\ T) &= \mathcal{Q}\ \text{demote}_*(T) \\ \text{demote}_*(\texttt{bool}) &= \texttt{bool} \\ \text{demote}_*(\texttt{nat}) &= \texttt{nat} \\ \text{demote}_*(\{\overline{x : \tau}\}) &= \left\{\overline{x : \text{demote}(\tau)}\right\} \\ \text{demote}_*(\texttt{type } t \texttt{ is } \overline{M}\ T) &= \text{demote}_*(T) \end{aligned}$$

$\boxed{\texttt{fields}(T) = \overline{x : \tau}}$ **Fields**

$$\texttt{fields}(T) = \begin{cases} \overline{x : \tau} & \text{if } T = \{\overline{x : \tau}\} \\ \texttt{fields}(T) & \text{if } T = \texttt{type } t \texttt{ is } \overline{M}\ T \\ \emptyset & \text{otherwise} \end{cases}$$

$\boxed{\texttt{update}(\Gamma, x, \tau)}$ **Type environment modification**

$$\texttt{update}(\Gamma, x, \tau) = \begin{cases} \Delta, x : \tau & \text{if } \Gamma = \Delta, x : \sigma \\ \Gamma & \text{otherwise} \end{cases}$$

$\boxed{\texttt{compat}(n, m, \mathcal{Q})}$ The relation $\texttt{compat}(n, m, \mathcal{Q})$ holds when the number of values sent, n, is compatible with the original number of values m, and the type quantity used, $\mathcal{Q}$.

$$\begin{aligned} \texttt{compat}(n, m, \mathcal{Q}) \Leftrightarrow & (\mathcal{Q} = \texttt{nonempty} \text{ and } n \geq 1) \text{ or} \\ & (\mathcal{Q} = \,! \text{ and } n = 1) \text{ or} \\ & (\mathcal{Q} = \texttt{empty} \text{ and } n = 0) \text{ or} \\ & (\mathcal{Q} = \texttt{every} \text{ and } n = m) \text{ or} \\ & \mathcal{Q} = \texttt{any} \end{aligned}$$

$\boxed{\texttt{values}(T) = \mathcal{V}}$ The function `values` gives a list of all of the values of a given base type.

$$\begin{aligned} \texttt{values}(\texttt{bool}) &= [\texttt{true}, \texttt{false}] \\ \texttt{values}(\texttt{nat}) &= [0, 1, 2, \ldots] \\ \texttt{values}(\texttt{list } T) &= [L | L \subseteq \texttt{values}(T), |L| < \infty] \\ \texttt{values}(\texttt{type } t \texttt{ is } \overline{M}\ T) &= \texttt{values}(T) \\ \texttt{values}(\{\overline{x : \mathcal{Q}\ T}\}) &= [\{\overline{x : \tau \mapsto v}\} | \overline{v \in \texttt{values}(T)}] \end{aligned}$$