1 Formalization

1.1 Syntax

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

$\mathcal{Q}, \mathcal{R}, \mathcal{S}$	::=	$! \mid \texttt{any} \mid \texttt{nonempty} \mid \texttt{empty} \mid \texttt{every}$	(type quantities)
M	::=	$\texttt{fungible} \mid \texttt{unique} \mid \texttt{immutable} \mid \texttt{consumable} \mid \texttt{asset}$	(type declaration modifiers)
T	::=	$\texttt{bool} \mid \texttt{nat} \mid \alpha \mid t[\overline{T}] \mid \texttt{table}(\overline{x})\ \{\overline{x : \tau}\}$	(base types)
τ, σ, π	::=	$\mathcal{Q}\ T$	(types)
T_V	::=	$\alpha\ \texttt{is}\ \overline{M}$	(type variable declaration)
$\mathcal{L}, \mathcal{K}$	::=	$\texttt{true} \mid \texttt{false} \mid n$	
	$\mid$	$x \mid \mathcal{L}.x \mid \texttt{var}\ x : T \mid [\overline{\mathcal{L}}] \mid \{\overline{x : \tau \mapsto \mathcal{L}}\}$	
	$\mid$	$\texttt{demote}(\mathcal{L}) \mid \texttt{copy}(\mathcal{L})$	
	$\mid$	$\mathcal{L}[\mathcal{L}] \mid \mathcal{L}[\mathcal{Q}\ \text{s.t.}\ f[\overline{T}](\overline{L})] \mid \texttt{consume}$	
`Decl`	::=	$\texttt{transformer}\ f[\overline{T_V}](\overline{x : \tau}) \to x : \tau\ \{\overline{\texttt{Stmt}}\}$	(transformers)
	$\mid$	$\texttt{type}\ t[\overline{T_V}]\ \texttt{is}\ \overline{M}\ T$	(type decl.)
`Trfm`	::=	$\texttt{new}\ t[\overline{T}](\overline{\mathcal{L}}) \mid f[\overline{T}](\overline{\mathcal{L}})$	(transformer calls)
`Stmt`	::=	$\mathcal{L} \to \mathcal{L} \mid \mathcal{L} \to \texttt{Trfm} \to \mathcal{L}$	(flows)
	$\mid$	$\texttt{try}\ \{\overline{\texttt{Stmt}}\}\ \texttt{catch}\ \{\overline{\texttt{Stmt}}\}$	(try-catch)
`Decl`	::=	$\texttt{transformer}\ f[\overline{T_V}](\overline{x : \tau}) \to x : \tau\ \{\overline{\texttt{Stmt}}\}$	(transformers)
`Prog`	::=	$\overline{\texttt{Decl}}; \overline{\texttt{Stmt}}$	(programs)

We use the following abbreviations

- $(\texttt{map}\ \tau \Rightarrow \sigma) := \texttt{table}(\texttt{key})\ \{\texttt{key} : \tau, \texttt{value} : \sigma\}$
- $(\texttt{list}\ \tau) := \texttt{table}(\texttt{idx})\ \{\texttt{idx} : \texttt{nat}, \texttt{value} : \tau\}$
- $\{\overline{x : \tau}\} := \texttt{table}(\cdot)\ \{\overline{x : \tau}\}$

[TODO: Need type formation rules to make sure things like types that are used are declared and that the key variables are a subset of the fields of a table]

1.2 Statics

Define $\# : \mathbb{N} \cup \{\infty\} \to \mathcal{Q}$ so that $\#(n)$ is the best approximation by type quantity of n, i.e.,

$$\#(n) = \begin{cases} \texttt{empty} & \text{if } n = 0 \\ ! & \text{if } n = 1 \\ \texttt{nonempty} & \text{if } n > 1 \\ \texttt{every} & \text{if } n = \infty \end{cases}$$

$\boxed{\Gamma \vdash (\mathcal{L} : \tau); u}$ **Locator Typing**

Here u is an *updater*, that is, $u \in \{\bot\} \cup ((\Gamma \times (\tau \to \tau)) \to \Gamma)$. The updater describes how the types of the values specified by the locator will be modified given some function $f : \tau \to \tau$. If $u = \bot$, that means that the updater cannot be applied—this typically **[atm, always]** means that the locator value(s) are immutable.

Define

$$u\|_T = \begin{cases} \bot & \text{if } T \texttt{ immutable} \\ u & \text{otherwise} \end{cases}$$

Let $\mathbf{1}_u$ be the identity updater, that is, $\mathbf{1}_u(\Delta, f) = \Delta$ for all Δ and f.

$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash b : !\ \texttt{bool}; \mathbf{1}_u}\ \textsc{Bool} \qquad \frac{}{\Gamma \vdash n : \#(n)\ \texttt{nat}; \mathbf{1}_u}\ \textsc{Nat}$$

[The idea is that both $\texttt{demote}(\mathcal{L})$ and $\texttt{copy}(\mathcal{L})$ give a demoted value, but $\texttt{demote}(\mathcal{L})$ gives a read-only value (so no copy needs to happen), whereas copy will actually copy all the data. The results below intentionally throw out the environment Δ, because we don't want to actually consume whatever references we used to get $\mathcal{L} : \tau$.]

$$\frac{\Gamma \vdash (\mathcal{L} : \tau); u}{\Gamma \vdash (\texttt{demote}(\mathcal{L}) : \text{demote}(\tau)); \bot}\ \textsc{Demote} \qquad \frac{\Gamma \vdash (\mathcal{L} : \tau); u}{\Gamma \vdash (\texttt{copy}(\mathcal{L}) : \text{demote}(\tau)); \mathbf{1}_u}\ \textsc{Copy}$$

[TODO: Finish adding the various updater functions as done in the Haskell file]

$$\frac{}{\Gamma, x : \mathcal{Q}\ T \vdash (x : \mathcal{Q}\ T); ((\Delta, f) \mapsto \Delta[x \mapsto f(\Delta(x))])\|_T}\ \textsc{Var}$$

$$\frac{\Gamma \vdash (\mathcal{L} : \mathcal{Q}\ !\ T); u \qquad (x : \tau) \in \text{fields}(T)}{\Gamma \vdash (\mathcal{L}.x : \tau); ((\Delta, f) \mapsto u(\Delta, \text{useField}_{T,x}(f)))\|_T}\ \textsc{Field}$$

where $\text{useField}_{T,x}(f)$ is defined by: **[Can we simplify this a little?] [Can/should define keys and fields functions so we can do more easily] [How ot keep track that a value used to be of some named typed????]**

$$\text{useField}_{T,x}(f) = \begin{cases} \texttt{table}(\overline{k})\ \{\overline{y : \tau}, x : f(\sigma), \overline{z : \pi}\} & \text{if } T = \texttt{table}(\overline{k})\ \{\overline{y : \tau}, x : \sigma, \overline{z : \pi}\} \\ \texttt{table}(\overline{k})\ \{\overline{y : \tau}, x : f(\sigma), \overline{z : \pi}\} & \text{if } \texttt{type}\ T\ \texttt{is}\ \overline{M}\ \texttt{table}(\overline{k})\ \{\overline{y : \tau}, x : \sigma, \overline{z : \pi}\} \end{cases}$$

$$\frac{\Gamma \vdash \overline{\mathcal{L} : \tau; u}}{\Gamma \vdash ([\overline{\mathcal{L}}] : \#(|\overline{\mathcal{L}}|)\ \texttt{list}\ \tau); (\Delta, f) \mapsto u_n(u_{n-1}(\cdots u_1(\Delta, f) \cdots, f), f)}\ \textsc{List}$$

$$\frac{\Gamma \vdash \overline{\mathcal{L} : \tau} \dashv \Delta}{\Gamma \vdash \{\overline{x : \tau \mapsto \mathcal{L}}\} : !\ \{\overline{x : \tau}\} \dashv \Delta}\ \textsc{Record}$$

[VarDef needs some way to get out of the definition so it can be used later? We could use the updater for this.]

$$\frac{}{\Gamma \vdash ((\texttt{var}\ x : T) : \texttt{empty}\ T); (\Delta, f) \mapsto \Delta[x \mapsto f(\texttt{empty}\ T)]}\ \textsc{VarDef} \qquad \frac{\tau\ \texttt{consumable}}{\Gamma \vdash \texttt{consume} : \tau}\ \textsc{Consume}$$

$$\frac{\Gamma \vdash (\mathcal{L} : \mathcal{Q}\ T); u \qquad \Gamma \vdash (\texttt{demote}(\mathcal{K}) : \text{demote}(\mathcal{R}\ T)); v}{\Gamma \vdash (\mathcal{L}[\mathcal{K}] : \mathcal{R}\ T); \texttt{select}(\mathcal{R}, u)}\ \textsc{Select}$$

where

$$\texttt{select}(\mathcal{Q}, u)(\Delta, f) = \begin{cases} \Delta & \text{if } \mathcal{Q} = \texttt{empty} \\ u(\Delta, f) & \text{if } \mathcal{Q} = \texttt{every} \\ u(\Delta, \tau \mapsto \tau \sqcup f(\tau)) & \text{otherwise} \end{cases}$$

$\boxed{\Gamma \vdash S \ \textbf{ok} \dashv \Delta}$ **Statement Well-formedness [TODO: Not sure that the final update call is right in Ok-Flow-Every]**

$$\frac{\Gamma \vdash \mathcal{L} : \mathcal{Q}\ T; u \qquad \Delta = u(\Gamma, (\mathcal{Q}'\ T') \mapsto \texttt{empty}\ T') \qquad \Delta \vdash \mathcal{K} : \mathcal{R}\ T; v}{\Gamma \vdash (\mathcal{L} \to \mathcal{K})\ \textbf{ok} \dashv v(\Delta, \tau \mapsto \tau \oplus \mathcal{Q})}\ \textsc{Ok-Flow-Every}$$

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

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

[Rule Ok-Flow-Transformer actually doesn't necessary add $\mathcal{Q}$ things to the destination because it's like a concat operation, need to either change that or change how much gets added.]

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

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

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

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

1.3 Dynamics

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

Definition 1. *An environment Σ is a tuple (μ, ρ) where μ : 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.

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

Locators.

$$\frac{}{\langle \Sigma, x \rangle \to \langle \Sigma, \text{storage}(\mu(x), \rho(\mu(x))) \rangle} \textsc{Loc-Id}$$

$$\frac{\ell \notin \texttt{dom}(\rho)}{\langle \Sigma, \texttt{var}\ x : T \rangle \to \langle \Sigma[\mu \mapsto \mu[x \mapsto \ell], \rho \mapsto \rho[\ell \mapsto []]], \ell \rangle} \textsc{Loc-VarDef}$$

$$\frac{\langle \Sigma, \mathcal{L} \rangle \to \langle \Sigma', \mathcal{L}' \rangle}{\langle \Sigma, \mathcal{L}.x \rangle \to \langle \Sigma', \mathcal{L}'.x \rangle} \textsc{Loc-Field-Congr} \qquad \frac{\rho(\ell) = \overline{k} \qquad \overline{\rho(k).x = j}}{\langle \Sigma, \ell.x \rangle \to \left\langle \Sigma', \overline{j} \right\rangle} \textsc{Loc-Field}$$

$$\frac{\langle \Sigma, \mathcal{L} \rangle \to \langle \Sigma', \mathcal{L}' \rangle}{\left\langle \Sigma, [\overline{\ell}, \mathcal{L}, \overline{\mathcal{L}'}] \right\rangle \to \left\langle \Sigma', [\overline{\ell}, \mathcal{L}', \overline{\mathcal{L}'}] \right\rangle} \textsc{Loc-List-Congr} \qquad \frac{k \notin \texttt{dom}(\rho)}{\left\langle \Sigma, [\overline{\ell}] \right\rangle \to \left\langle \Sigma[\rho \mapsto \rho[k \mapsto \overline{\ell}]], k \right\rangle} \textsc{Loc-List}$$

$$\frac{\langle \Sigma, \mathcal{L} \rangle \to \langle \Sigma'', \mathcal{L}'' \rangle}{\langle \Sigma, \mathcal{L}[\mathcal{L}'] \rangle \to \langle \Sigma'', \mathcal{L}''[\mathcal{L}'] \rangle} \textsc{Loc-Val-Src-Congr} \qquad \frac{\langle \Sigma, \mathcal{L}' \rangle \to \langle \Sigma'', \mathcal{L}'' \rangle}{\langle \Sigma, \ell[\mathcal{L}'] \rangle \to \langle \Sigma'', \ell[\mathcal{L}'] \rangle} \textsc{Loc-Val-Sel-Congr}$$

$$\frac{}{\left\langle \Sigma, \overline{\ell}[\overline{k}] \right\rangle \to \left\langle \Sigma, \texttt{select}(\rho, \overline{\ell}, \overline{k}) \right\rangle} \textsc{Loc-Val}$$

$$\texttt{select}(\rho, \overline{\ell}, \overline{k}) = \begin{cases} [] & \text{if } \overline{k} = [] \\ \texttt{revert} & \text{if } \overline{\ell} = [] \text{ and } \overline{k} \neq [] \\ j, \texttt{select}(\rho, \overline{\ell} \setminus j, \overline{k'}) & \text{if } \overline{k} = (i, \overline{k'}) \text{ and } j \in \overline{\ell} \text{ and } \rho(i) = \rho(j) \\ \texttt{select}(\rho, \overline{\ell}, \overline{k'}) & \text{if } \overline{k} = (i, \overline{k'}) \text{ and } \forall j \in \overline{\ell}. \rho(i) \neq \rho(j) \end{cases}$$

[TODO Finish this rule/figuring out exactly how all this reference stuff works...]

$$\frac{}{\langle \Sigma, \ell \to k \rangle \to \langle \Sigma[\rho \mapsto \rho[\ell \mapsto \rho(\ell) \backslash, k \mapsto \rho(k) + \rho(\ell)]] \rangle} \textsc{Flow}$$

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

$$\frac{\texttt{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \qquad \rho'(\ell) = \mathcal{V} \qquad \rho'(\mu'(x)) = \mathcal{W} \qquad \mathcal{W} \leq \mathcal{V}}{\left\langle \Sigma, \mathcal{S} \xrightarrow{x} \mathcal{D} \right\rangle \to \langle \Sigma'[\rho \mapsto \rho'[\ell \mapsto \mathcal{V} - \mathcal{W}]], \texttt{put}(\mathcal{W}, \mathcal{D}) \rangle} \textsc{Flow-Var}$$

$$\frac{\texttt{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \qquad \rho'(\ell) = \mathcal{V} \qquad \rho'(\mu'(x)) = \mathcal{W} \qquad \mathcal{W} \not\leq \mathcal{V}}{\left\langle \Sigma, \mathcal{S} \xrightarrow{x} \mathcal{D} \right\rangle \to \langle \Sigma', \texttt{revert} \rangle} \textsc{Flow-Var-Fail}$$

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

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

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

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

[NOTE: It is important that we flow an empty list in the Flow-Transformer-Done rule, otherwise we may fail to allocate a variable as expected.]

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

We introduce a new statement, $\texttt{try}(\Sigma, \overline{S_1}, \overline{S_2})$, 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 $\texttt{revert}$.

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

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

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

[TODO: Need to be sure to handle uniqueness correctly]

1.4 Auxiliaries

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

$\boxed{\text{isAsset}(\overline{T_V}, \tau)}$ **Asset Types**

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

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

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

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

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

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

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

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 $\textbf{dom}(\Gamma)$ *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*

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

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 $\Gamma \sqcup \Delta$.

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

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

$\boxed{\text{modifiers}(\overline{T_V}, T) = \overline{M}}$ **Type Modifiers**

$$\text{modifiers}(\overline{T_V}, T) = \begin{cases} \overline{M} & \text{if } T = \texttt{type}\ t\ \texttt{is}\ \overline{M}\ T' \\ \overline{M} & \text{if } (T\ \texttt{is}\ \overline{M}) \in \overline{T_V} \\ \emptyset & \text{otherwise} \end{cases}$$

$\boxed{\text{demote}(\tau) = \sigma}$ $\boxed{\text{demote}_*(T_1) = T_2}$ **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.

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

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

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

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

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

$\boxed{\text{compat}(n, m, \mathcal{Q})}$ The relation 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, $\mathcal{Q}$.

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

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

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