

1 Formalization

1.1 Syntax

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

We use the following abbreviations

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

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

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

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

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

Define

$$u \parallel_T = \begin{cases} \perp & \text{if } T \text{ immutable} \\ u & \text{otherwise} \end{cases}$$

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

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

[The idea is that both $\text{demote}(\mathcal{L})$ and $\text{copy}(\mathcal{L})$ give a demoted value, but $\text{demote}(\mathcal{L})$ 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 $\mathcal{L} : \tau$.]

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

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

$$\frac{}{\Gamma, x : \mathcal{Q} T \vdash (x : \mathcal{Q} T); ((\Delta, f) \mapsto \Delta[x \mapsto f(\Delta(x))]) \parallel_T} \text{Var} \\ \frac{\Gamma \vdash (\mathcal{L} : \mathcal{Q} ! T); u \quad (x : \tau) \in \text{fields}(T)}{\Gamma \vdash (\mathcal{L}.x : \tau); ((\Delta, f) \mapsto u(\Delta, \text{useField}_{T,x}(f))) \parallel_T} \text{Field}$$

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????]

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

$$\frac{\Gamma \vdash \overline{\mathcal{L} : \tau}; u}{\Gamma \vdash ([\overline{\mathcal{L}}] : \#([\overline{\mathcal{L}}]) \text{ list } \tau); (\Delta, f) \mapsto u_n(u_{n-1}(\dots u_1(\Delta, f) \dots, f), f)} \text{List} \\ \frac{\Gamma \vdash \overline{\mathcal{L} : \tau} \dashv \Delta}{\Gamma \vdash \{x : \tau \mapsto \overline{\mathcal{L}}\} : ! \{ \overline{x : \tau} \} \dashv \Delta} \text{Record}$$

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

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

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

where

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

$\boxed{\Gamma \vdash S \text{ ok} \vdash \Delta}$ **Statement Well-formedness** [TODO: Not sure that the final update call is right in OK-FLOW-EVERY]

$$\frac{\Gamma \vdash \mathcal{L} : \mathcal{Q} T; u \quad \Delta = u(\Gamma, (\mathcal{Q}' T') \mapsto \text{empty } T') \quad \Delta \vdash \mathcal{K} : \mathcal{R} T; v \quad u \neq \perp \quad v \neq \perp}{\Gamma \vdash (\mathcal{L} \rightarrow \mathcal{K}) \text{ ok} \vdash v(\Delta, \tau \mapsto \tau \oplus \mathcal{Q})} \text{OK-FLOW-EVERY}$$

$$\frac{\Gamma \vdash \mathcal{L} : \mathcal{Q} T_1 \vdash \Delta \quad \text{typeof}(f, \overline{T}) = (\overline{x} : \overline{\sigma}, y : \text{elemtype}(T_1)) \rightarrow z : \mathcal{R} T_2 \{ \overline{\text{Stmt}} \} \quad \forall i. \text{demote}(\Gamma(x_i)) = \sigma_i \quad \text{update}(\Delta, \mathcal{L}, \Delta(\mathcal{L}) \ominus \mathcal{Q}) \vdash \mathcal{K} : \mathcal{S} T_2 \vdash \Xi}{\Gamma \vdash (\mathcal{L} \rightarrow f[\overline{T}](\overline{x}) \rightarrow \mathcal{K}) \text{ ok} \vdash \text{update}(\Xi, \mathcal{K}, \Xi(\mathcal{K}) \oplus \mathcal{Q})} \text{OK-FLOW-TRANSFORMER}$$

$$\frac{\Gamma \vdash \overline{S}_1 \text{ ok} \vdash \Delta \quad \Gamma \vdash \overline{S}_2 \text{ ok} \vdash \Xi}{\Gamma \vdash (\text{try } \{\overline{S}_1\} \text{ catch } \{\overline{S}_2\}) \text{ ok} \vdash \Delta \sqcup \Xi} \text{OK-TRY}$$

[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.]

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

$$\frac{\overline{T}_V, \overline{x} : \overline{\tau} \vdash \overline{\text{Stmt}} \text{ ok} \vdash \Gamma, y : \sigma \quad \forall \pi \in \text{img}(\Gamma). \neg \text{isAsset}(\overline{T}_V, \pi)}{\vdash (\text{transformer } f[\overline{T}_V](\overline{x} : \overline{\tau}) \rightarrow y : \sigma \{ \overline{\text{Stmt}} \}) \text{ ok}} \text{OK-TRANSFORMER}$$

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

$$\frac{\vdash \overline{\text{Decl}} \text{ ok} \quad \emptyset \vdash \overline{\text{Stmt}} \text{ ok} \vdash \Gamma \quad \forall \tau \in \text{img}(\Gamma). \neg \text{isAsset}(\emptyset, \tau)}{(\overline{\text{Decl}}; \overline{\text{Stmt}}) \text{ ok}} \text{OK-PROG}$$

1.3 Dynamics

$$\begin{aligned} V &::= \text{true} \mid \text{false} \mid n \mid \{ \overline{x} : \overline{\tau} \mapsto n \} \\ \mathcal{V} &::= \overline{V} \\ \text{Stmt} &::= \dots \mid \text{revert} \mid \text{try}(\Sigma, \overline{\text{Stmt}}, \overline{\text{Stmt}}) \end{aligned}$$

Definition 1. An environment Σ is a tuple (μ, ρ) where $\mu : \text{IDENTIFIERNAMES} \rightarrow \mathbb{N}$ is the variable lookup environment, and $\rho : \mathbb{N} \rightarrow \mathcal{V}$ is the storage environment.

$$\boxed{\langle \Sigma, \overline{\text{Stmt}} \rangle \rightarrow \langle \Sigma, \overline{\text{Stmt}} \rangle}$$

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

$$\frac{\langle \Sigma, S_1 \rangle \rightarrow \langle \Sigma', \overline{S}_3 \rangle}{\langle \Sigma, S_1 \overline{S}_2 \rangle \rightarrow \langle \Sigma', \overline{S}_3 \overline{S}_2 \rangle} \text{SEQ} \quad \frac{}{\langle \Sigma, (\text{revert}) \overline{S} \rangle \rightarrow \langle \Sigma, \text{revert} \rangle} \text{REVERT}$$

Locators.

$$\begin{array}{c}
\frac{}{\langle \Sigma, x \rangle \rightarrow \langle \Sigma, \text{storage}(\mu(x), \rho(\mu(x))) \rangle} \text{Loc-Id} \\
\\
\frac{\ell \notin \text{dom}(\rho)}{\langle \Sigma, \text{var } x : T \rangle \rightarrow \langle \Sigma[\mu \mapsto \mu[x \mapsto \ell], \rho \mapsto \rho[\ell \mapsto []]], \ell \rangle} \text{Loc-VarDef} \\
\\
\frac{\langle \Sigma, \mathcal{L} \rangle \rightarrow \langle \Sigma', \mathcal{L}' \rangle}{\langle \Sigma, \mathcal{L}.x \rangle \rightarrow \langle \Sigma', \mathcal{L}'.x \rangle} \text{Loc-Field-Congr} \qquad \frac{\rho(\ell) = \bar{k} \quad \overline{\rho(k).x = j}}{\langle \Sigma, \ell.x \rangle \rightarrow \langle \Sigma', \bar{j} \rangle} \text{Loc-Field} \\
\\
\frac{\langle \Sigma, \mathcal{L} \rangle \rightarrow \langle \Sigma', \mathcal{L}' \rangle}{\langle \Sigma, [\bar{\ell}, \mathcal{L}, \bar{\mathcal{L}}'] \rangle \rightarrow \langle \Sigma', [\bar{\ell}, \mathcal{L}', \bar{\mathcal{L}}'] \rangle} \text{Loc-List-Congr} \qquad \frac{k \notin \text{dom}(\rho)}{\langle \Sigma, [\bar{\ell}] \rangle \rightarrow \langle \Sigma[\rho \mapsto \rho[k \mapsto \bar{\ell}], k] \rangle} \text{Loc-List} \\
\\
\frac{\langle \Sigma, \mathcal{L} \rangle \rightarrow \langle \Sigma'', \mathcal{L}'' \rangle}{\langle \Sigma, \mathcal{L}[\mathcal{L}'] \rangle \rightarrow \langle \Sigma'', \mathcal{L}''[\mathcal{L}'] \rangle} \text{Loc-Val-Src-Congr} \qquad \frac{\langle \Sigma, \mathcal{L}' \rangle \rightarrow \langle \Sigma'', \mathcal{L}'' \rangle}{\langle \Sigma, \ell[\mathcal{L}'] \rangle \rightarrow \langle \Sigma'', \ell[\mathcal{L}'] \rangle} \text{Loc-Val-Sel-Congr} \\
\\
\frac{}{\langle \Sigma, \bar{\ell}[\bar{k}] \rangle \rightarrow \langle \Sigma, \text{select}(\rho, \bar{\ell}, \bar{k}) \rangle} \text{Loc-Val}
\end{array}$$

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

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

$$\frac{}{\langle \Sigma, \ell \rightarrow k \rangle \rightarrow \langle \Sigma[\rho \mapsto \rho[\ell \mapsto \rho(\ell) \setminus k \mapsto \rho(k) + \rho(\ell)]] \rangle} \text{Flow}$$

$$\begin{array}{c}
\frac{\text{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell)}{\langle \Sigma, \mathcal{S} \rightarrow \mathcal{D} \rangle \rightarrow \langle \Sigma'[\rho \mapsto \rho'[\ell \mapsto []], \text{put}(\rho'(\ell), \mathcal{D})] \rangle} \text{FLOW-EVERY} \\
\\
\frac{\text{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \quad \rho'(\ell) = \mathcal{V} \quad \rho'(\mu'(x)) = \mathcal{W} \quad \mathcal{W} \leq \mathcal{V}}{\langle \Sigma, \mathcal{S} \xrightarrow{x} \mathcal{D} \rangle \rightarrow \langle \Sigma'[\rho \mapsto \rho'[\ell \mapsto \mathcal{V} - \mathcal{W}], \text{put}(\mathcal{W}, \mathcal{D})] \rangle} \text{FLOW-VAR} \\
\\
\frac{\text{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \quad \rho'(\ell) = \mathcal{V} \quad \rho'(\mu'(x)) = \mathcal{W} \quad \mathcal{W} \not\leq \mathcal{V}}{\langle \Sigma, \mathcal{S} \xrightarrow{x} \mathcal{D} \rangle \rightarrow \langle \Sigma', \text{revert} \rangle} \text{FLOW-VAR-FAIL} \\
\\
\frac{\text{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \quad \rho'(\ell) = \mathcal{V} \quad \mathcal{U} = [v \in \mathcal{V} \mid \langle \Sigma', f(\bar{x}, v) \rangle \rightarrow^* \langle \Sigma'', k \rangle \text{ and } \rho''(k) = \text{true}] \quad \text{compat}(|\mathcal{U}|, |\mathcal{V}|, \mathcal{Q})}{\langle \Sigma, \mathcal{S} \xrightarrow{\mathcal{Q} \text{ s.t. } f(\bar{x})} \mathcal{D} \rangle \rightarrow \langle \Sigma'[\rho' \mapsto \rho'[\ell \mapsto \rho'(\ell) - \mathcal{U}], \text{put}(\mathcal{U}, \mathcal{D})] \rangle} \text{FLOW-FILTER} \\
\\
\frac{\text{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \quad \rho'(\ell) = \mathcal{V} \quad \mathcal{U} = [v \in \mathcal{V} \mid \langle \Sigma', f(\bar{x}, v) \rangle \rightarrow^* \langle \Sigma'', k \rangle \text{ and } \rho''(k) = \text{true}] \quad \neg\text{compat}(|\mathcal{U}|, |\mathcal{V}|, \mathcal{Q})}{\langle \Sigma, \mathcal{S} \xrightarrow{\mathcal{Q} \text{ s.t. } f(\bar{x})} \mathcal{D} \rangle \rightarrow \langle \Sigma', \text{revert} \rangle} \text{FLOW-FILTER-FAIL} \\
\\
\frac{\text{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \quad \rho'(\ell) = v, \mathcal{V} \quad \langle \Sigma'[\rho \mapsto \rho'[\ell \mapsto \mathcal{V}]], f(\bar{x}, v) \rangle \rightarrow^* \langle \Sigma'', k \rangle}{\langle \Sigma, \mathcal{S} \rightarrow f(\bar{x}) \rightarrow \mathcal{D} \rangle \rightarrow \langle (\mu', \rho''), \text{put}([\rho''(k)], \mathcal{D}) (\mathcal{S} \rightarrow f(\bar{x}) \rightarrow \mathcal{D}) \rangle} \text{FLOW-TRANSFORMER} \\
\\
\frac{\text{resolve}(\Sigma, \mathcal{S}) = (\Sigma', \ell) \quad \rho'(\ell) = []}{\langle \Sigma, \mathcal{S} \rightarrow f(\bar{x}) \rightarrow \mathcal{D} \rangle \rightarrow \langle \Sigma, \text{put}([], \mathcal{D}) \rangle} \text{FLOW-TRANSFORMER-DONE}
\end{array}$$

[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.]

$$\frac{\ell \notin \text{dom}(\rho) \quad \text{transformer } f(\overline{y} : \overline{\tau}) \rightarrow z : \sigma \{ \overline{S} \} \quad \mu' = \overline{y} \mapsto \overline{\mu(x)}, z \mapsto \ell}{\langle \Sigma, f(\bar{x}) \rangle \rightarrow \langle (\mu', \rho[\ell \mapsto []]), \overline{S} \ell \rangle} \text{CALL}$$

We introduce a new statement, $\text{try}(\Sigma, \overline{S}_1, \overline{S}_2)$, 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 `revert`.

$$\begin{array}{c}
\overline{\langle \Sigma, \text{try } \{ \overline{S}_1 \} \text{ catch } \{ \overline{S}_2 \} \rangle} \rightarrow \overline{\langle \Sigma, \text{try}(\Sigma, \overline{S}_1, \overline{S}_2) \rangle} \text{TRY-START} \\
\\
\frac{\langle \Sigma, \overline{S}_1 \rangle \rightarrow \langle \Sigma'', \overline{S}_1' \rangle}{\langle \Sigma, \text{try}(\Sigma', \overline{S}_1, \overline{S}_2) \rangle \rightarrow \langle \Sigma'', \text{try}(\Sigma', \overline{S}_1', \overline{S}_2) \rangle} \text{TRY-STEP} \\
\\
\overline{\langle \Sigma, \text{try}(\Sigma', \text{revert}, \overline{S}_2) \rangle} \rightarrow \overline{\langle \Sigma', \overline{S}_2 \rangle} \text{TRY-REVERT} \quad \overline{\langle \Sigma, \text{try}(\Sigma', \cdot, \overline{S}_2) \rangle} \rightarrow \Sigma \text{TRY-DONE}
\end{array}$$

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

1.4 Auxiliaries

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

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

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

τ consumable **Consumable Types**

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

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

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

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

Define the operator \ominus , called split, as the unique function $\mathbf{Quant}^2 \rightarrow \mathbf{Quant}$ such that

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

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

Definition 4. We can consider a type environment Γ as a function $\text{IDENTIFIERS} \rightarrow \text{TYPES} \cup \{\perp\}$ as follows:

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

We write $\text{dom}(\Gamma)$ to mean $\{x \in \text{IDENTIFIERS} \mid \Gamma(x) \neq \perp\}$, 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

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

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 $\Gamma \sqcup \Delta$.

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

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

$$\boxed{\text{modifiers}(\overline{T}_V, T) = \overline{M}} \quad \textbf{Type Modifiers}$$

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

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

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.

$$\text{demote}(\mathcal{Q} T) = \mathcal{Q} \text{demote}_*(T)$$

$$\text{demote}_*(\text{bool}) = \text{bool}$$

$$\text{demote}_*(\text{nat}) = \text{nat}$$

$$\text{demote}_*(\{\overline{x} : \overline{\tau}\}) = \{\overline{x} : \text{demote}(\overline{\tau})\}$$

$$\text{demote}_*(\text{type } t \text{ is } \overline{M} T) = \text{demote}_*(T)$$

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

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

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

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

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

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

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

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