1 Specification

1.1 Syntax

$C \in \textsc{ContractNames}$ $\quad$ $m \in \textsc{TransactionNames}$

$t \in \textsc{TypeNames}$ $\quad$ $x, y, z \in \textsc{Identifiers}$

$n \in \mathbb{Z}$

$\mathfrak{q}$	::=	$! \mid \texttt{any} \mid \texttt{nonempty}$	(selector quantifiers)
$\mathcal{Q}, \mathcal{R}, \mathcal{S}$	::=	$\mathfrak{q} \mid \texttt{empty} \mid \texttt{every}$	(type quantities)
$\mathcal{C}$	::=	$\texttt{option} \mid \texttt{set} \mid \texttt{list}$	(collection type constructors)
T	::=	$\texttt{void} \mid \texttt{bool} \mid \texttt{nat} \mid \mathcal{C}\ \tau \mid \tau \rightsquigarrow \tau \mid \{\overline{x : \tau}\} \mid t$	(base types)
τ, σ, π	::=	$\mathcal{Q}\ T$	(types)
$\mathcal{V}$	::=	$n \mid \texttt{true} \mid \texttt{false} \mid \texttt{emptyval} \mid \lambda x : \tau . E$	(values)
$\mathcal{L}$	::=	$x \mid x[x] \mid x.x$	(locations)
E	::=	$\mathcal{V} \mid \mathcal{L} \mid x.m(\overline{x}) \mid \texttt{some}(x) \mid s\ \texttt{in}\ x \mid \{\overline{x : \tau \mapsto x}\}$	
	$\mid$	$\texttt{let}\ x : \tau := E\ \texttt{in}\ E \mid \texttt{if}\ x\ \texttt{then}\ E\ \texttt{else}\ E$	(expressions)
s	::=	$\mathcal{L} \mid \texttt{everything} \mid \mathfrak{q}\ x : \tau\ \textbf{s.t.}\ E$	(selector)
$\mathcal{S}$	::=	$\mathcal{L} \mid \texttt{new}\ t$	(sources)
$\mathcal{D}$	::=	$\mathcal{L} \mid \texttt{consume}$	(destinations)
F	::=	$\mathcal{S} \xrightarrow{s} x \to \mathcal{D}$	(flows)
$\texttt{Stmt}$	::=	$F \mid E \mid \texttt{revert}(E) \mid \texttt{try}\ S\ \texttt{catch}(x : \tau)\ S \mid \texttt{if}\ x\ \texttt{then}\ S\ \texttt{else}\ S$	
	$\mid$	$\texttt{var}\ x : \tau := E \mid S; S \mid \texttt{pack} \mid \texttt{unpack}(x)$	(statements)
M	::=	$\texttt{fungible} \mid \texttt{nonfungible} \mid \texttt{consumable} \mid \texttt{asset}$	(type declaration modifiers)
$\texttt{Decl}$	::=	$x : \tau$	(field)
	$\mid$	$\texttt{type}\ t\ \texttt{is}\ \overline{M}\ T$	(type declaration)
	$\mid$	$[\texttt{private}]\ \texttt{transaction}\ m(\overline{x : \tau})\ \texttt{returns}\ x : \tau\ \texttt{do}\ S$	(transactions)
	$\mid$	$\texttt{view}\ m(\overline{x : \tau})\ \texttt{returns}\ \tau := E$	(views)
	$\mid$	$\texttt{on create}(\overline{x : \tau})\ \texttt{do}\ S$	(constructor)
$\texttt{Con}$	::=	$\texttt{contract}\ C\ \{\ \overline{\texttt{Decl}}\ \}$	(contracts)
$\texttt{Prog}$	::=	$\overline{\texttt{Con}}\ ;\ S$	(programs)

Figure 1: Abstract syntax of LANGUAGE-NAME.

[We have public and private transactons...we could also have a public/private type?]

In the surface language, "collection types" (i.e., $\mathcal{Q}\ \mathcal{C}\ \tau$ or a transformer) are by default `any`, but all other types, like `nat`, are !.

[Some simplification ideas] [Could get rid of selecting by locations and only allowed selecting with quantifiers, and just optimize things like $! \ x : \tau$ **s.t.** $x = y$ **into a lookup. Also, allowing any type quantity in a selector lets us do away with everything. Would actually be even nicer if we allowed any type quantity to appear in the quantifier, because then we wouldn't even need a special rule for everything.] [We could also get rid of "if" and instead do something like** any $x : \tau$ **s.t.** if b then $x = y$ else $false$**]**

[Contract types should be consumable assets by default (consuming a contract is a self-destruct?)]

1.2 Statics

$\Gamma, \Delta, \Xi \quad ::= \quad \emptyset \mid \Gamma, x : \tau \quad$ (type environments)

Definition 1. *Define* $\boldsymbol{Quant} = \{\mathbf{empty}, \mathbf{any}, !, \mathbf{nonempty}, \mathbf{every}\}$, *and call any* $\mathcal{Q} \in \boldsymbol{Quant}$ *a* type quantity. *Define* $\mathbf{empty} < \mathbf{any} < ! < \mathbf{nonempty} < \mathbf{every}$.

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

[The syntax for record "fields" and type environments is the same...could just use it]

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

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

$$(\mathcal{Q}\ T)\ \texttt{consumable} \iff \texttt{consumable} \in \texttt{modifiers}(T) \text{ or } \neg((\mathcal{Q}\ T)\ \texttt{asset})$$

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

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

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

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

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 3. *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 4. *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*

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

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

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

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

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

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

$\boxed{\Gamma \vdash E : \tau \dashv \Delta}$ **Expression Typing**

These rules are for ensuring that expressions are well-typed, and keeping track of which variables are used throughout the expression. Most rules do **not** change the context, with the notable exceptions of internal calls and record-building operations. We begin with the rules for typing the various literal forms of values.

$$\frac{}{\Gamma \vdash \texttt{emptyval} : \texttt{empty}\ \mathcal{C}\ \tau \dashv \Gamma}\ \textsc{Empty-Val} \qquad \frac{\Gamma, x : \tau \vdash E : \sigma \dashv \Gamma}{\Gamma \vdash (\lambda x : \tau . E) : \texttt{every list}\ !\ (\tau \rightsquigarrow \sigma) \dashv \Gamma}\ \textsc{Transformer}$$

$$\frac{\Gamma \vdash x : \tau \dashv \Delta}{\Gamma \vdash \texttt{some}(x) : !\ \texttt{option}\ \tau \dashv \Delta}\ \textsc{Some} \qquad \frac{\Gamma \vdash \overline{y : \tau} \dashv \Delta}{\Gamma \vdash \{\overline{x : \tau \mapsto y}\} \dashv \Delta}\ \textsc{Build-Rec}$$

Next, the lookup rules. Notably, the Demote-Lookup rule allows the use of variables of an asset type in an expression without consuming the variable as Lin-Lookup does. However, it is still safe, because it is treated as its demoted type, which is always guaranteed to be a non-asset.

$$\frac{}{\Gamma, x : \tau \vdash x : \text{demote}(\tau) \dashv \Gamma, x : \tau}\ \textsc{Demote-Lookup} \qquad \frac{}{\Gamma, x : \mathcal{Q}\ T \vdash x : \mathcal{Q}\ T \dashv \Gamma, x : \texttt{empty}\ T}\ \textsc{Lin-Lookup}$$

$$\frac{\Gamma \vdash x : \{\overline{y : \tau}\} \vdash \Gamma \qquad f : \sigma \in \overline{y : \tau}}{\Gamma \vdash x.f : \sigma \vdash \Gamma}\ \textsc{Record-Field-Lookup}$$

[Record field lookup rule doesn't take into account that fields can store assets...]

The expression s `in` x allows checking whether a flow will succeed without the EAFP-style ("Easier to ask for forgiveness than permission"; e.g., Python). A flow $A \xrightarrow{s} B$ is guaranteed to succeed when "s `in` A" is true and "s `in` B" is false.

$$\frac{\Gamma \vdash x\ \texttt{stores}_\mathcal{Q}\ \tau \qquad \Gamma \vdash s\ \texttt{selects}\ \text{demote}(\tau)}{\Gamma \vdash (s\ \texttt{in}\ x) : \texttt{bool} \dashv \Gamma}\ \textsc{Check-In}$$

We distinguish between three kinds of calls: view, internal, and external. A view call is guaranteed to not change any state in the receiver, while both internal and external calls may do so. The difference between internal and external calls is that we may transfer assets to an internal call, but **not** to an external call, because we cannot be sure any external contract will properly manage the

asset of our contract.

$$\frac{\texttt{typeof}(C, m) = \{\overline{a : \tau}\} \rightsquigarrow \sigma \qquad \Gamma, x : C \vdash \overline{y : \tau} \dashv \Gamma, x : C}{\Gamma, x : C \vdash x.m(\overline{y}) : \sigma \dashv \Gamma, x : C} \textsc{View-Call}$$

$$\frac{\texttt{dom}(\texttt{fields}(C)) \cap \texttt{dom}(\Gamma) = \emptyset \qquad \texttt{typeof}(C, m) = \{\overline{a : \tau}\} \rightsquigarrow \sigma \qquad \Gamma, \texttt{this} : C \vdash \overline{y : \tau} \dashv \Delta, \texttt{this} : C}{\Gamma, \texttt{this} : C \vdash \texttt{this}.m(\overline{y}) : \sigma \dashv \Delta, \texttt{this} : C} \textsc{Internal-Tx-Call}$$

$$\frac{\texttt{dom}(\texttt{fields}(C)) \cap \texttt{dom}(\Gamma) = \emptyset \qquad (\texttt{transaction}\ m(\overline{a : \tau})\ \texttt{returns}\ \sigma\ \texttt{do}\ S) \in \texttt{decls}(D) \qquad \Gamma, \texttt{this} : C, x : D \vdash \overline{y : \tau} \dashv \Gamma, \texttt{this} : C, x : D}{\Gamma, \texttt{this} : C, x : D \vdash x.m(\overline{y}) : \sigma \dashv \Gamma, \texttt{this} : C, x : D} \textsc{External-Tx-Call}$$

$$\frac{(\texttt{on create}(\overline{x : \tau})\ \texttt{do}\ S) \in \texttt{decls}(C) \qquad \Gamma \vdash \overline{y : \tau} \dashv \Gamma}{\Gamma \vdash \texttt{new}\ C(\overline{y}) : C} \textsc{New-Con}$$

Finally, the rules for If and Let expressions. In Let-Expr, we ensure that the newly bound variable is either consumed or is not an asset in the body.

$$\frac{\Gamma \vdash x : \texttt{bool} \dashv \Gamma \qquad \Gamma \vdash E_1 : \tau \dashv \Delta \qquad \Gamma \vdash E_2 : \tau \dashv \Xi}{\Gamma \vdash (\texttt{if}\ x\ \texttt{then}\ E_1\ \texttt{else}\ E_2) : \tau \dashv \Delta \sqcup \Xi} \textsc{If-Expr}$$

$$\frac{\Gamma \vdash E_1 : \tau \dashv \Delta \qquad \Delta, x : \tau \vdash E_2 : \pi \dashv \Xi, x : \sigma \qquad \neg(\sigma\ \texttt{asset})}{\Gamma \vdash (\texttt{let}\ x : \tau := E_1\ \texttt{in}\ E_2) : \pi \dashv \Xi} \textsc{Let-Expr}$$

$\boxed{\Gamma \vdash \mathcal{S}\ \texttt{provides}_{\mathcal{Q}}\ \tau}$ **Source Typing**

$$\frac{}{\Gamma, S : \tau \vdash S\ \texttt{provides}_{!}\ \tau} \textsc{Provide-One} \qquad \frac{}{\Gamma, S : \mathcal{Q}\ \mathcal{C}\ \tau \vdash S\ \texttt{provides}_{\mathcal{Q}}\ \tau} \textsc{Provide-Col}$$

$$\frac{(\texttt{type}\ t\ \texttt{is}\ \overline{M}\ T) \in \texttt{decls}(C)}{\Gamma, \texttt{this} : C \vdash (\texttt{new}\ t)\ \texttt{provides}_{\texttt{every}}\ !\ t} \textsc{Provide-Source}$$

[Note, it will be too difficult to implement to make every kind of selector work with the sources, because the quantified selector can contain arbitrary expressions. It needs to be restricted somehow; the current rules only ensure you don't flow everything from a source. Could write special Flow-Source rules.]

$\boxed{\Gamma \vdash \mathcal{D}\ \texttt{accepts}\ \tau}$ **Destination Typing** **[Prevent variables that are supposed to store exactly one of something from receiving another?]** Note that the type quantities in Accept-One are different on the left and right of the turnstile. This is because, for example, when I have D : `nonempty set nat`, it is reasonable to flow into it some S : `any set nat`.

$$\frac{}{\Gamma, D : \mathcal{Q}\ T \vdash S\ \texttt{accepts}\ \mathcal{R}\ T} \textsc{Accept-One} \qquad \frac{}{\Gamma, D : \mathcal{Q}\ \mathcal{C}\ \tau \vdash S\ \texttt{accepts}\ \tau} \textsc{Accept-Col}$$

$$\frac{\tau\ \texttt{consumable}}{\Gamma \vdash \texttt{consume accepts}\ \tau} \textsc{Accept-Consume}$$

$\boxed{\Gamma \vdash s\ \texttt{selects}_{\mathcal{Q}}\ \tau}$ **Selectors**

$$\frac{\Gamma \vdash \mathcal{L} : \mathcal{Q}\ T \dashv \Gamma}{\Gamma \vdash \mathcal{L}\ \texttt{selects}_{\mathcal{Q}}\ \mathcal{Q}\ T}\ \textsc{Select-Loc} \qquad \frac{\Gamma \vdash x : \mathcal{Q}\ \mathcal{C}\ \tau \dashv \Gamma}{\Gamma \vdash x\ \texttt{selects}_{\mathcal{Q}}\ \tau}\ \textsc{Select-Col}$$

$$\frac{}{\Gamma \vdash \texttt{everything}\ \texttt{selects}_{\texttt{every}}\ \tau}\ \textsc{Select-Everything} \qquad \frac{\Gamma, x : \tau \vdash p : \textbf{bool} \dashv \Gamma, x : \tau}{\Gamma \vdash (\mathfrak{q}\ x : \tau\ \textbf{s.t.}\ p)\ \texttt{selects}_{\mathfrak{q}}\ \tau}\ \textsc{Select-Quant}$$

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

[In the new flow rule, we always use a transformer. However, that just means we desugar something like $A \xrightarrow{s} B$ into $A \xrightarrow{s} (\lambda x : \tau.x) \to B$. In the real compiler, this can be optimized.]

Flows are the main construct for transferring resources. A flow has four parts: a source, a selector, a transformer, and a destination. The selector acts as a function that "chooses" part of the source's resources to flow. These resources then get applied to the transformer, which is an applicative functor applied to a function type. **[Bringing back one would let us do all the collections the same way in all of these flow-related rules, which would be nice.]**

(Non)Ambiguity of Flow Rules Consider the flow $A \xrightarrow{s} f \to B$. **[Actually, the type of f will probably be enough to distinguish the cases, but if we want to desguar the flows into flows containing a transformer always then we would have to infer its type and run into the same issue again.]** The only way that the choice of which Provide, Select, or Accept rules could be ambiguous **[I think...]** is if A and B are both collections containing the same type, and either s is a collection containing the same type or it is `everything`. If A, B, and s are all collections containing the same type, then we could use either version of the rules (the appropriate One rule or the appropriate Col rule). However, regardless of the rule we choose, the outcome will be the same. For example, if we use the Select-Loc rule, A will now store the quantity `any` (unless s is `empty`), which is correct, because we don't know how many values will be transferred by s. Finally, if s is `everything`, the same argument applies—the outcome will be the same regardless of which rule we pick.

$$\frac{\begin{array}{c}\Gamma \vdash A\ \texttt{provides}_{\mathcal{Q}}\ \tau \qquad \Gamma \vdash s\ \texttt{selects}_{\mathcal{R}}\ \tau \qquad \min(\mathcal{Q}, \mathcal{R}) < \texttt{every} \\ \Delta = \texttt{update}(\Gamma, A, \Gamma(A) \ominus \mathcal{R}) \qquad \Delta \vdash f : \tau \rightsquigarrow \sigma \dashv \Delta \qquad \Delta \vdash B\ \texttt{accepts}\ \sigma\end{array}}{\Gamma \vdash (A \xrightarrow{s} f \to B)\ \textbf{ok} \dashv \texttt{update}(\Delta, B, \Delta(B) \oplus \min(\mathcal{Q}, \mathcal{R}))}\ \textsc{Ok-Flow}$$

[TODO: Finish handling currying transformers.]

$$\frac{\Gamma \vdash E : \tau \dashv \Delta \qquad \Delta, x : \tau \vdash S \ \mathbf{ok} \dashv \Xi, x : \sigma \qquad \neg(\sigma \ \mathtt{asset})}{\Gamma \vdash (\mathtt{var}\ x : \tau := E\ \mathtt{in}\ S)\ \mathbf{ok} \dashv \Xi}\ \textsc{Ok-Var-Def}$$

$$\frac{\Gamma \vdash x : \mathtt{bool} \dashv \Gamma \qquad \Gamma \vdash S_1 \ \mathbf{ok} \dashv \Delta \qquad \Gamma \vdash S_2 \ \mathbf{ok} \dashv \Xi}{\Gamma \vdash (\mathtt{if}\ x\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2)\ \mathbf{ok} \dashv \Delta \sqcup \Xi}\ \textsc{Ok-If}$$

$$\frac{\Gamma \vdash S_1 \ \mathbf{ok} \dashv \Delta \qquad \Gamma, x : \tau \vdash S_2 \ \mathbf{ok} \dashv \Xi, x : \sigma \qquad \neg(\sigma \ \mathtt{asset})}{\Gamma \vdash (\mathtt{try}\ S_1\ \mathtt{catch}\ (x : \tau)\ S_2)\ \mathbf{ok} \dashv \Delta \sqcup \Xi}\ \textsc{Ok-Try}$$

$$\frac{\Gamma \vdash E : \tau \dashv \Gamma}{\Gamma \vdash \mathtt{revert}(E)\ \mathbf{ok} \dashv \Gamma}\ \textsc{Ok-Revert} \qquad \frac{\Gamma \vdash E : \tau \dashv \Delta \qquad \neg(\tau \ \mathtt{asset})}{\Gamma \vdash E \ \mathbf{ok} \dashv \Delta}\ \textsc{Ok-Expr}$$

$$\frac{\Gamma \vdash S_1 \ \mathbf{ok} \dashv \Delta \qquad \Delta \vdash S_2 \ \mathbf{ok} \dashv \Xi}{\Gamma \vdash (S_1; S_2)\ \mathbf{ok} \dashv \Xi}\ \textsc{Ok-Seq}$$

$$\frac{\mathtt{this}.f : \tau \in \mathtt{fields}(C)}{\Gamma, \mathtt{this} : C \vdash \mathtt{unpack}(f)\ \mathbf{ok} \dashv \Gamma, \mathtt{this} : C, \mathtt{this}.f : \tau}\ \textsc{Ok-Unpack}$$

$$\frac{(\Gamma|_{\mathtt{dom}(\mathtt{fields}(C))}) \leq \mathtt{fields}(C) \qquad \Delta = \{x : \tau \in \Gamma \mid x \notin \mathtt{dom}(\mathtt{fields}(C))\}}{\Gamma, \mathtt{this} : C \vdash \mathtt{pack}\ \mathbf{ok} \dashv \Delta, \mathtt{this} : C}\ \textsc{Ok-Pack}$$

$\boxed{\vdash_C \mathtt{Decl}\ \mathbf{ok}}$ **Declaration Well-formedness**

$$\frac{\Gamma = \mathtt{this} : C, \mathtt{fields}(C), \overline{x : \tau} \qquad \Gamma \vdash E : \sigma \dashv \Gamma}{\vdash_C (\mathtt{view}\ m(\overline{x : \tau})\ \mathtt{returns}\ \sigma := E)\ \mathbf{ok}}\ \textsc{Ok-View}$$

$$\frac{\begin{array}{c}\mathtt{this} : C, \overline{x : \tau}, y : \mathtt{empty}\ T \vdash S \ \mathbf{ok} \dashv \Delta, \mathtt{this} : C, y : \mathcal{Q}\ T \\ \mathtt{dom}(\mathtt{fields}(C)) \cap \mathtt{dom}(\Delta) = \emptyset \qquad \forall x : \tau \in \Delta. \neg(\tau \ \mathtt{asset}) \qquad \neg(\mathcal{Q}\ T \ \mathtt{asset})\end{array}}{\vdash_C (\mathtt{transaction}\ m(\overline{x : \tau})\ \mathtt{returns}\ y : \mathcal{Q}\ T\ \mathtt{do}\ S)\ \mathbf{ok}}\ \textsc{Ok-Tx-Public}$$

$$\frac{\begin{array}{c}\mathtt{this} : C, \overline{x : \tau}, y : \mathtt{empty}\ T \vdash S \ \mathbf{ok} \dashv \Delta, \mathtt{this} : C, y : \mathcal{Q}\ T \\ \mathtt{dom}(\mathtt{fields}(C)) \cap \mathtt{dom}(\Delta) = \emptyset \qquad \forall x : \tau \in \Delta. \neg(\tau \ \mathtt{asset})\end{array}}{\vdash_C (\mathtt{private\ transaction}\ m(\overline{x : \tau})\ \mathtt{returns}\ y : \mathcal{Q}\ T\ \mathtt{do}\ S)\ \mathbf{ok}}\ \textsc{Ok-Tx-Private}$$

A field definition is always okay, as long as the type doesn't have the `every` modifier. **[Add this restriction to the rest of the places where we write types.] [Maybe we should always restrict variable definitions so that you can only write named types that appear in the current contract. This isn't strictly necessary, because everything will still work, but you'll simply never be able to get a value of an asset type not created in the current contract.]**

$$\frac{\mathcal{Q} \neq \mathtt{every}}{\vdash_C (x : \mathcal{Q}\ T)\ \mathbf{ok}}\ \textsc{Ok-Field}$$

A type declaration is okay as long as it has the `asset` modifier if its base type is an asset. Note that this restriction isn't necessary, but is intended to help users realize which types are assets without unfolding the entire type definition.

$$\frac{T\ \mathtt{asset} \implies \mathtt{asset} \in \overline{M}}{\vdash_C (\mathtt{type}\ t\ \mathtt{is}\ \overline{M}\ T)\ \mathbf{ok}}\ \textsc{Ok-Type}$$

Note that we need to have constructors, because only the contract that defines a named type is allowed to create values of that type, and so it is not always possible to externally initialize all contract fields.

$$\frac{\texttt{fields}(C) = \overline{\texttt{this}.f : \mathcal{Q}\ T} \qquad \texttt{this} : C, \overline{x : \tau}, \overline{\texttt{this}.f : \texttt{empty}\ T} \vdash S\ \textbf{ok} \dashv \Delta \qquad \forall y : \sigma \in \Delta. \neg(\sigma\ \texttt{asset})}{\vdash_C (\texttt{on create}(\overline{x : \tau})\ \texttt{do}\ S)\ \textbf{ok}}\ \textsc{Ok-Constructor}$$

$\boxed{\texttt{Con}\ \textbf{ok}}$ **Contract Well-formedness**

$$\frac{\forall d \in \overline{\texttt{Decl}}. (\vdash_C d\ \textbf{ok}) \qquad \exists! d \in \overline{\texttt{Decl}}. \exists \overline{x : \tau}, S. d = \texttt{on create}(\overline{x : \tau})\ \texttt{do}\ S}{(\texttt{contract}\ C\ \{\overline{\texttt{Decl}}\})\ \textbf{ok}}\ \textsc{Ok-Con}$$

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

$$\frac{\forall C \in \overline{\texttt{Con}}. C\ \textbf{ok} \qquad \emptyset \vdash S \dashv \emptyset}{(\overline{\texttt{Con}}; S)\ \textbf{ok}}\ \textsc{Ok-Prog}$$

Other Auxiliary Definitions **[Eliminate all the locations except for x and then use flows to extract and put stuff back?]** $\boxed{\texttt{modifiers}(T) = \overline{M}}$ **Type Modifiers**

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

$\boxed{\text{demote}(\tau) = \sigma}$ $\boxed{\text{demote}_*(T_1) = T_2}$ **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. **[TODO: Transformer demotion? My current thought is we should split functions and transformers, with the latter being able to "hold" a resource after being partially applied, and therefore being able to be an asset. Alternatively, can just not allow for currying...]**

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

$\boxed{\texttt{decls}(C) = \overline{\texttt{Decl}}}$ **Contract Declarations**

$$\texttt{decls}(C) = \overline{\texttt{Decl}} \text{ where } (\texttt{contract}\ C\ \{\overline{\texttt{Decl}}\})$$

$\boxed{\texttt{fields}(C) = \Gamma}$ **Contract Fields**

$$\texttt{fields}(C) = \{\texttt{this}.f : \tau \mid f : \tau \in \texttt{decls}(C)\}$$

$\boxed{\texttt{typeof}(C, m) = \tau \rightsquigarrow \sigma}$ **Method Type Lookup**

$$\texttt{typeof}(C, m) = \begin{cases} \{\overline{x : \tau}\} \rightsquigarrow \sigma & \text{if } (\texttt{private transaction } m(\overline{x : \tau}) \texttt{ returns } y : \sigma \texttt{ do } S) \in \texttt{decls}(C) \\ \{\overline{x : \tau}\} \rightsquigarrow \sigma & \text{if } (\texttt{transaction } m(\overline{x : \tau}) \texttt{ returns } y : \sigma \texttt{ do } S) \in \texttt{decls}(C) \\ \{\overline{x : \tau}\} \rightsquigarrow \sigma & \text{if } (\texttt{view } m(\overline{x : \tau}) \texttt{ returns } \sigma := E) \in \texttt{decls}(C) \end{cases}$$

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

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

[Asset retention theorem?] [Resource accessiblity?]
[What guarantees should we provide (no errors except for flowing a resource that doesn't exist in the source/already exists in the destination)?]

2 Introduction

LANGUAGE-NAME is a DSL for implementing programs which manage resources, targeted at writing smart contracts.

2.1 Contributions

We make the following main contributions:

- **Safety guarantees**: similar to **[or maybe just exactly]** linear types**[or maybe uniqueness types? need to read more about this]**, preventing accidental resource loss or duplication. Additionally, provides some amount of reentrancy safety.
  - We can evaluate these by formalizing the language and proving them; the formalization is something that would be nice to do anyway.
- **Simplicity**: The language is quite simple—it makes writing typical smart contract programs easier and shorter, because many common pitfalls in Solidity are automatically handled by the language, such as overflow/underflow, checking of balances, short address attacks, etc.
  - We can evaluate these by comparing LOC, cyclomatic complexity, etc. Not sure what the right metric would be. **[Or how cyclomatic complexity would work exactly in this language.]**
  - We can also evaluate via a user study, but that will take a long(er) time.
- **[Optimizations?]** Some of the Solidity contracts are actually inefficient because:
  1. They use lots of modifiers which repeat checks (see reference implementation of ERC-721).
  2. They tend to use arrays to represent sets. Maybe this is more efficient for very small sets, but checking containment is going to be much faster with a `mapping (X => bool)` eventually.
  - We can evaluate this by profiling or a simple opcode count (which is not only a proxy for performance, but also means that deploying the contract will be cheaper).

3 Language Intro

The basic state-changing construct in the language is a *flow*. A flow describes a transfer of a *resource* from one *storage* to another. A *transaction* is a sequence of statements.

Each flow has a *source*, a *destination*, and a *selector*. The source and destination are two storages which hold a resource, and the selector describes which part of the resource in the source should be transferred to the destination. A flow may optionally have a *name*.

Note that all flows fail if they can't be performed. For example, a flow of fungible resources fails if there is enough of the resource, and a flow of a nonfungible resource fails if the selected value doesn't exist in the source location.

NOTE: If we wanted to be "super pure", we can implement preconditions with just flows by doing something like:

```
{ contractCreator = msg.sender } --[ true ]-> consume
```

This works because `{ contractCreator = msg.sender } : set bool` (specifically, a singleton), so if `contractCreator = msg.sender` doesn't evaluate to true, then we will fail to consume true from it. **[I don't think actually doing this is a good idea; at least, not in the surface language. Maybe it would simplify the compiler and/or formalization, but it's interesting/entertaining.]**

The DAO attack We can prevent the DAO attack (the below is from `https://consensys.github.io/smart-contract-best-practices/known_attacks/`):

```
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    // At this point, the caller's code is executed, and can call withdrawBalance again
    require(msg.sender.call.value(amountToWithdraw)(""));
    userBalances[msg.sender] = 0;
}
```

In LANGUAGE-NAME, we would write this as:

```
transaction withdrawBalance():
    userBalances[msg.sender] --> msg.sender.balance
```

Not only is this simpler, but the compiler can automatically place the actual call that does the transfer last, meaning that the mistake could simply never be made.