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)
$\mathcal{V}$	::=	$n \mid \texttt{true} \mid \texttt{false} \mid \texttt{empty} \mid \lambda x : \tau . E$	(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)
$\mathcal{N}$	::=	$\mathcal{L} \mid \texttt{new}\ C(\overline{x}) \mid \texttt{new}\ t \mid \texttt{consume}$	(nodes)
F	::=	$\mathcal{N} \xrightarrow{s} \mathcal{N}$	(flows)
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)
$\mathcal{D}$	::=	$x : \tau$	(field)
	$\mid$	$\texttt{type}\ t\ \texttt{is}\ \overline{M}\ T$	(type declaration)
	$\mid$	$\texttt{transaction}\ m(\overline{x : \tau})\ \texttt{returns}\ x : \tau\ \texttt{do}\ S$	(transactions)
	$\mid$	$\texttt{view}\ m(\overline{x : \tau})\ \texttt{returns}\ \tau := E$	(views)
	$\mid$	$\texttt{on create}(\overline{x : \tau})\ \texttt{do}\ S$	(constructor)
Con	::=	$\texttt{contract}\ C\ \{\ \overline{\mathcal{D}}\ \}$	(contracts)
Prog	::=	$\overline{\texttt{Con}}\ ;\ S$	(programs)

Figure 1: Abstract syntax of LANGUAGE-NAME.

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?)]

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

1.2 Statics

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]

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

[It should be the case that a transformer can have an output of an asset type if and only if it has an input asset type (and it the case of curried transformers, that some input type is an asset).]

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

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

Definition 2. *Let* $\mathcal{Q}, \mathcal{R} \in \boldsymbol{Quant}$. *Define the commutative operator* $\oplus$, *called* combine, *as the function:*

$$\begin{aligned}\oplus : \textsc{Type} \times \boldsymbol{Quant} &\to \textsc{Type} \\ (\mathcal{Q}\ T, \mathcal{R}) &\mapsto \max(\mathcal{Q}, \mathcal{R})\ T\end{aligned}$$

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

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

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 $(\mathcal{Q} \ominus \mathcal{R})\ T$. $\mathcal{Q} \oplus \mathcal{R}$ 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. *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.

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

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

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.

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

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

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.

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

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. Similarly, we cannot be sure that a view of an external contract **really** is a

view, and so we can only call views of the current contract.

$$\frac{(\texttt{view}\ m(\overline{a:\tau})\ \texttt{returns}\ \sigma := E) \in \texttt{decls}(C) \qquad \Gamma, \texttt{this} : C \vdash \overline{y:\tau} \dashv \Gamma, x : C}{\Gamma, \texttt{this} : C \vdash \texttt{this}.m(\overline{y}) : \sigma \dashv \Gamma, \texttt{this} : C}\ \textsc{View-Call}$$

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

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

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.

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

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

$\boxed{\Gamma \vdash \mathcal{S} :: \tau \Rrightarrow_{\mathcal{Q}} \sigma}$ **Storage Typing** The syntax $\tau \Rrightarrow_{\mathcal{Q}} \sigma$ means that the storage accepts τ and provides σ; that is, you can flow τ into it, and when you flow out of it, you get σ; moreover, you get at most $\mathcal{Q}$ of them.

$$\frac{}{\Gamma, S : \mathcal{Q}\ T \vdash S :: \mathcal{R}\ T \Rrightarrow_{!} \mathcal{Q}\ T}\ \textsc{Store-One} \qquad \frac{}{\Gamma, S : \mathcal{Q}\ \mathcal{C}\ \tau \vdash S :: \tau \Rrightarrow_{\mathcal{Q}} \tau}\ \textsc{Store-Col}$$

$$\frac{}{\Gamma, x : \mathcal{Q}\ (\tau \rightsquigarrow \sigma) \vdash x :: \tau \Rrightarrow_{\mathcal{Q}} \sigma}\ \textsc{Store-Transformer} \qquad \frac{\tau\ \texttt{consumable}}{\Gamma \vdash \texttt{consume} :: \tau \Rrightarrow_{\texttt{empty}} \texttt{void}}\ \textsc{Store-Consume}$$

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

$$\frac{(\texttt{type}\ t\ \texttt{is}\ \overline{M}\ T) \in \texttt{decls}(C)}{\Gamma, \texttt{this} : C \vdash \texttt{new}\ t :: \texttt{void} \Rrightarrow_{\texttt{every}}\ t}\ \textsc{Store-Source}$$

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash A :: \tau \Rrightarrow_{\mathcal{Q}} \sigma \qquad \Gamma \vdash s \ \mathtt{selects}_{\mathcal{R}} \ \mathrm{demote}(\sigma) \\ \Delta = \mathtt{update}(\Gamma, A, \Gamma(A) \ominus \mathcal{R}) \qquad \Delta \vdash B :: \sigma \Rrightarrow_S \pi \qquad \min(\mathcal{Q}, \mathcal{R}) < \mathtt{every}\end{array}}{\Gamma \vdash (A \xrightarrow{s} B) \ \mathbf{ok} \dashv \mathtt{update}(\Delta, B, \Delta(B) \oplus \min(\mathcal{Q}, \mathcal{R}))} \ \textsc{Wf-Flow}$$

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

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

$$\frac{\Gamma \vdash S_1 \ \mathbf{ok} \dashv \Delta \qquad \Gamma, x : \tau \vdash S_2 \ \mathbf{ok} \dashv \Xi, x : \tau}{\Gamma \vdash (\mathtt{try}\ S_1\ \mathtt{catch}\ (x : \tau)\ S_2) \ \mathbf{ok} \dashv \Delta \sqcup \Xi} \ \textsc{Wf-Try} \qquad \frac{\Gamma \vdash E : \tau \dashv \Gamma}{\Gamma \vdash \mathtt{revert}(E) \ \mathbf{ok} \dashv \Gamma} \ \textsc{Wf-Revert}$$

$$\frac{\Gamma \vdash E : \tau \dashv \Delta \qquad \neg(\tau \ \mathtt{asset})}{\Gamma \vdash E \ \mathbf{ok} \dashv \Delta} \ \textsc{Wf-Expr} \qquad \frac{\Gamma \vdash S_1 \ \mathbf{ok} \dashv \Delta \qquad \Delta \vdash S_2 \ \mathbf{ok} \dashv \Xi}{\Gamma \vdash (S_1; S_2) \ \mathbf{ok} \dashv \Xi} \ \textsc{Wf-Seq}$$

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

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

$\boxed{\vdash_C \mathcal{D} \ \mathbf{ok}}$ **Declaration Well-formedness**

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

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

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

[Public transaction shouldn't be able to return an asset? Actually, everything would be a lot nicer if you just couldn't move assets around with transactions. Probably wouldn't be that limiting, because of the demotion system.]

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.

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

$\boxed{\mathtt{Con}\ \mathbf{ok}}$ **Contract Well-formedness**

$$\frac{\forall d \in \overline{\mathcal{D}}. (\vdash_C d \ \mathbf{ok}) \qquad \exists! d \in \overline{\mathcal{D}}. \exists \overline{x : \tau}, S. d = \mathtt{on}\ \ \mathtt{create}(\overline{x : \tau})\ \mathtt{do}\ S}{(\mathtt{contract}\ C\ \{\overline{\mathcal{D}}\}) \ \mathbf{ok}} \ \textsc{Wf-Con}$$

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

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

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**

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

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

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

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

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

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

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

[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.

4 Random Thoughts

[Ignore this for now, just some stuff for fun that may or may not end up being useful.] [At some point, maybe for antoher language, would like to figure out the more general type quantity thing where you can freely combine `set` and `list` and `nonempty` and `one`, etc.] [We could also add sum types and "sum type quantities" so you can say `x : Left (a + b)`. Something like $\{\ell, r\}$ where ℓ and r are state specifiers. Probably would work best with labeled sum types.] [Add at "at most one" quantity, which makes the type quantities into a group?] [Free group of quantities list set and option makes teh type quantities we care about? What are the inverses?]

$$\texttt{nonempty} = \texttt{option}^{-1}$$

Definition 5. ***[Don't think this really has a good name]*** *A* resource $\mathcal{R}$ *is a tuple* $(R, +, 0, \leq, -)$ *where*

(i) $(R, +, 0)$ *is a monoid.*

(ii) $(R, \leq)$ *is a partial order that is compatible with* $(R, +, 0)$*. That is, for any* $x, y \in R$ *such that* $x \leq y$*, and for any* $z \in R$*, we have* $x + z \leq y + z$*.*

(iii) $- : R \times R \to R$ *is a function so that for any* $x, y \in R$ *such that* $y \leq x$*, we have* $(x - y) + y = x$*.*

Examples

1. The natural numbers with the standard operations is a resource $(\mathbb{N}, +, 0, \leq, -)$, where $-$ is *saturating subtraction*: $n - m = 0$ if $m > n$.
2. For any set A, we can build the resource $(\mathcal{P}(A), \cup, \emptyset, \subseteq, \setminus)$, where $X \setminus Y$ is the set difference operation.
3. Similarly, given any set A, we can build the resource $(\mathcal{P}_{\text{fin}}(A), \cup, \emptyset, \subseteq, \setminus)$; this resource is called the *nonfungible resource on* A, written $\text{nf}(A)$.
4. **[I have no idea if this is useful, but it's fun]** The set of strings on an alphabet Σ can be made into the *prefix resource* $(\Sigma^*, \varepsilon, \cdot, \leq_p, -_p)$ or the *suffix resource* $(\Sigma^*, \varepsilon, \cdot, \leq_s, -_s)$, where $\cdot$ is concatenation, $x \leq_p y$ if x is a prefix of y and similarly $x \leq_s y$ if x is a suffix of y. The functions $-_p$ and $-_s$ are defined as follows:
$$(x \cdot y) -_p x = y$$

and

$$(x \cdot y) -_s y = x$$

Definition 6. *For any two resources $\mathcal{R}$ and $\mathcal{S}$, their* direct product, *$\mathcal{R} \oplus \mathcal{S}$ is $(R \times S, +_{RS}, (0_R, 0_S), \leq_{RS}, -_{RS})$, where*

$$\begin{aligned}(r_1, s_1) +_{RS} (r_2, s_2) &:= (r_1 +_R r_2, s_1 +_S s_2)\\ (r_1, s_1) -_{RS} (r_2, s_2) &:= (r_1 -_R r_2, s_1 -_S s_2)\\ (r_1, s_1) \leq_{RS} (r_2, s_2) &: \iff r_1 \leq_R r_2 \wedge s_1 \leq_S s_2\end{aligned}$$

Definition 7. *Let $\mathcal{R}$ be a resource and let A be a set. The exponential resource $\mathcal{R}^A$* ***[I suspect this would be an exponential, anyway]*** *is defined so*

$$\mathcal{R}^A := (R^A, +, \mathbf{0}, \leq, -)$$

where for $f, g \in R^A$, we define

$$\begin{aligned}\mathbf{0}(x) &:= 0\\ (f + g)(x) &:= f(x) + g(x)\\ f \leq g &:\Leftrightarrow \forall x \in A. f(x) \leq g(x)\\ (f - g)(x) &:= f(x) - g(x)\end{aligned}$$

Note that $f - g$ is only defined when $g \leq f$.