LANGUAGE-NAME: A DSL for the Safe Blockchain Assets

Reed Oei
reedoei2@illinois.edu
University of Illinois at Urbana-Champaign
Urbana, USA

ACM Reference Format:
Reed Oei. 2020. LANGUAGE-NAME: A DSL for the Safe Blockchain Assets. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

[Authors/affiliations?] LANGUAGE-NAME is a DSL for implementing programs which manage assets, targeted at writing smart contracts.

1.1 Contributions

We make the following contributions with LANGUAGE-NAME.

- **Safety guarantees**: LANGUAGE-NAME ensures that assets are properly managed, eliminating reuse and asset-loss bugs.
- **Flow abstraction**: LANGUAGE-NAME uses a new abstraction called a *flow* to encode semantic information about a program into the code.
- **Conciseness**: LANGUAGE-NAME makes writing typical smart contract programs more concise by handling common pitfalls automatically.

[Potential benefits of the language. Some of these are already discussed in the paper.

- **Good expression of financial assets: fungible, nonfungible/general uniqueness constraints, consumable vs. nonconsumable. NOTE: These are things that Obsidian doesn't express automatically. Emphasize the uniqueness stuff is actually not any more difficult/inefficient than existing solutions.**
- **Atomic flow construct encodes semantic intent**
- **Maybe the language is efficient, but would need an implementation to evaluate this.**
- **Flows are interesting?**

]

2 LANGUAGE DESCRIPTION

A LANGUAGE-NAME program is made of many *contracts*, each containing *declarations*, such as *transactions*, *views*, *types*, and *fields*. A contract is a high-level unit of functionality, which behaves **["very similarly"]** to a contract in Solidity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Example

```
contract EIP20 {
    type Token is fungible asset uint256
    accounts : map address => Token
    transaction transfer(dst : address, amount : uint256):
        account[msg.sender] --[ amount ]-> account[dst]
}
```

Figure 1: An implement of ERC-20's transfer function in LANGUAGE-NAME.

2.2 Flows

The LANGUAGE-NAME language is built around the concept of a *flow*, an atomic, state-changing operation describing the transfer of a asset. Each flow has at least a *source* and a *destination*; they may optionally have a *selector* or a *transformer*, which default to **`everything`** and the identity transformer, respectively.

There are two special kinds of assets: *fungible* and *nonfungible*. **[I think there's also assets which are neither fungible nor nonfungible.] [Not sure about these definitions.]** A *fungible* assets are those whose values are not unique and can be combined: for example, ERC-20 tokens are fungible, because two accounts may have the same number of tokens—the number isn't the token, but instead describes **how many** tokens there are. A *nonfungible* asset is an asset that is unique and immutable, and can be held in at most one location. For example, ERC-721 **[cite]** (discussed in more depth in Section 3.2) tokens are nonfungible—each token is unique and can be held by at most one account at a time. LANGUAGE-NAME dynamically ensures that all newly created nonfungible assets are unique, and statically ensures that the resources are not duplicated or changed. **[This dynamic checking is no more costly that the standard approaches used for this purpose, should we discuss this?]** Furthermore, it supports data structures that make working with assets easier, such as *linkings*, a bidirectional mapping between keys and a collection of values, with special operations to support modeling of "token accounts" (i.e., addresses which have a balance consisting of a set of tokens.

The source and destination of a flow are two storages which *provide* and *accept* assets, and the selector describes which subpart of the value of the asset(s) in the source should be transferred to the destination. If not present, all assets will be transferred.

All flows fail if the selected assets are not present in the source, or if the selected assets cannot be added to the destination. For example, a flow of fungible assets fails if there is not enough of the asset in the soruce, and a flow of a nonfungible asset fails if the selected value doesn't exist in the source location.

$$\begin{array}{llll}
\mathfrak{q} & ::= & ! \mid \texttt{any} \mid \texttt{nonempty} & \text{(selector quantifiers)} \\
Q & ::= & \mathfrak{q} \mid \texttt{empty} \mid \texttt{every} & \text{(type quantities)} \\
T & ::= & \texttt{bool} \mid \texttt{nat} \mid \texttt{map}\ \tau \Rightarrow \sigma \mid t \mid \ldots & \text{(base types)} \\
\tau & ::= & Q\ T & \text{(types)} \\
\mathcal{V} & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \texttt{emptyval} \mid \ldots & \text{(values)} \\
\mathcal{L} & ::= & x \mid x.x & \text{(locations)} \\
E & ::= & \mathcal{V} \mid \mathcal{L} \mid \texttt{total}\ t \mid \ldots & \text{(expressions)} \\
s & ::= & \mathcal{L} \mid \texttt{everything} \mid \mathfrak{q}\ x : \tau\ \textbf{s.t.}\ E & \text{(selector)} \\
\mathcal{S} & ::= & \mathcal{L} \mid \texttt{new}\ t & \text{(sources)} \\
\mathcal{D} & ::= & \mathcal{L} \mid \texttt{consume} & \text{(destinations)} \\
F & ::= & \mathcal{S} \xrightarrow{s} \mathcal{D} & \text{(flows)} \\
\texttt{Stmt} & ::= & F \mid \texttt{Stmt}; \texttt{Stmt} \mid \ldots & \text{(statements)} \\
M & ::= & \texttt{fungible} \mid \texttt{nonfungible} & \\
 & \mid & \texttt{consumable} \mid \texttt{asset} & \text{(type modifiers)} \\
\texttt{Decl} & ::= & \texttt{type}\ t\ \texttt{is}\ \overline{M}\ T & \text{(type declaration)} \\
 & \mid & \texttt{transaction}\ m(\overline{x : \tau})\ \texttt{returns}\ x : \tau\ \texttt{do}\ \texttt{Stmt} & \text{(transactions)} \\
 & \mid & \ldots & \\
\texttt{Con} & ::= & \texttt{contract}\ C\ \{\ \overline{\texttt{Decl}}\ \} & \text{(contracts)}
\end{array}$$

Figure 2: A fragment of the abstract syntax of the core calculus of LANGUAGE-NAME.

2.3 Syntax

Figure 6 shows a fragment of the syntax of the core calculus of LANGUAGE-NAME, which uses A-normal form **[cite]** and makes several other simplifications to the surface LANGUAGE-NAME language. These simplifications are performed automatically by the compiler. **[TODO: We have formalized this core calculus (in K???).]**

3 CASE STUDIES

3.1 ERC-20

[Cite all Solidity code properly]

Figure 3 shows implementations of the ERC-20 **[cite]** standard in both Solidity and LANGUAGE-NAME, one of the most commonly implemented standards on the Ethereum blockchain **[cite]**. Only the core functions of transfer, transferFrom, and approve are shown, with the exception of totalSupply in the LANGUAGE-NAME implementation (included because to show the use of the **total** operator). All event code has been omitted, because LANGUAGE-NAME handles events in the same way as Solidity. This contract shows several advantages of the flow abstraction:

- **Precondition checking**: For a flow to succeed, the source must have enough assets and the destination must be capable of receiving the assets flowed. In this case, the balance of the sender must be greater than the amount sent, and the balance of the destination must not overflow when it receives the tokens. Code checking these two conditions is automatically inserted, ensuring that the checks cannot be forgotten.
- **Data-flow tracking**: It is clear where the resources are flowing from the code itself, which may not be apparent in more complicated implementations, such as those involving transfer fees. Furthermore, developers must explicitly mark all times that assets are *consumed*, and only assets marked as consumable may be consumed. This restriction prevents, in this example, tokens from being consumed, and can also be used to ensure that other assets, like ether, are not consumed.
- **Error messages**: When a flow fails, LANGUAGE-NAME provides **[TODO: **will provide**]** automatic, descriptive error messages, such as "Cannot flow '<amount>' Token from account[<src>] to **[Not sure exactly what the error message should be.]** The default implementation provides no error message forcing developers to write their own. Flows enable the generation of the messages by encoding the semantic information of a **transfer** into the program, instead of using low-level incrementing and decrementing.

3.2 ERC-721

[Another benefit here is that linkings are a good datastructure for accounts of nonfungible assets. Of course, you could always implement a linking in a Solidity library... However, you still wouldn't have the property that only one account is guaranteed to hold each token, because of the nonfungibility/uniqueness.]

The ERC-721 standard **[cite]** requires many invariants hold: the tokens must be unique, at most one non-owning account can have "approval" for a token, we must be able to support "operators" who can manage all of the tokens of a user, among others. Because LANGUAGE-NAME is designed to handle assets, it has features to help developers ensure that these correctness properties hold. A LANGUAGE-NAME implementation has several benefits: because of the asset abstraction, we can be sure that token references will not be duplicated or lost; because Token has been declared as **nonfungible**, we can be sure that we will not mint two of the same token.

Figure 4 shows an implementation ERC-721's transferFrom function in both Solidity and LANGUAGE-NAME. The Solidity implementation is extracted from one of the reference implementations of ERC-721 given on its official Ethereum EIP page. In addition to the invariant required by the specification, there are also internal invariant which the contract must maintain, such as the connection between idToOwner and ownerToNFTokenCount, which are handled by LANGUAGE-NAME. This example demonstrates the benefits of having nonfungible assets and linkings built into the language itself.

3.3 Voting

[Solidity impl. comes from "Solidity by Example" page]

[Can include this section if we don't only want to talk about tokens...] The **nonfungible** modifier is useful in many programs, even those not dealing with financial assets, like ERC-721 contracts. We can also use **nonfungible** to remove certain incorrect behaviors from a voting contract, shown in Figure 5.

3.4 The DAO attack

[Not sure how notable this is.] [Describe 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
```

```
contract EIP20 {
    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;
    function transferFrom(address from, address to, uint256 value)
        public returns (bool success) {
        require(balances[from] >= value &&
                allowed[from][msg.sender] >= value);
        balances[to] += value;
        balances[from] -= value;
        allowed[from][msg.sender] -= value;
        return true;
    }
    function approve(address spender, uint256 value)
        public returns (bool success) {
        allowed[msg.sender][spender] = value;
        return true;
    }
}
```

```
contract EIP20 {
    type Token is fungible asset uint256
    type Approval is fungible consumable asset uint256
    accounts : map address => Token
    allowances : map address => map address => Approval
    transaction transferFrom(src : address, dst : address, amount : uint256):
        allowances[src][dst] --[ amount ]-> consume
        account[src] --[ amount ]-> account[dst]
    transaction approve(dst : address, amount : uint256):
        allowances[msg.sender][dst] --> consume
        new Approval --[ amount ]-> allowances[msg.sender][dst]
}
```

Figure 3: A Solidity and a LANGUAGE-NAME implementation of the core functions of the ERC-20 standard.

```
    // 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
```

Because of the additional information encoded in the flow construct, the compiler can output the safe version of the above code—reducing the balance before peforming the external call—without any user intervention.

4 DISCUSSION

5 RELATED WORK

[?]

6 CONCLUSION

A FORMALIZATION

A.1 Syntax

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

In the surface language, "collection types" (i.e., $Q\ C\ \tau$ 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?)]

A.2 Statics

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

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

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

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

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

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

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

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

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

```
contract NFToken {
  mapping (uint256 => address) idToOwner;
  mapping (uint256 => address) idToApproval;
  mapping (address => uint256) ownerToNFTokenCount;
  mapping (address => mapping (address => bool)) ownerToOperators;
  modifier canTransfer(uint256 _tokenId) {
    address tokenOwner = idToOwner[_tokenId];
    require (tokenOwner == msg.sender ||
             idToApproval[_tokenId] == msg.sender ||
             ownerToOperators[tokenOwner][msg.sender],
             NOT_OWNER_APPROWED_OR_OPERATOR);
    _;
  }
  modifier validNFToken(uint256 _tokenId) {
    require (idToOwner[_tokenId] != address (0), NOT_VALID_NFT);
    _;
  }
  function transferFrom(address _from, address _to, uint256 _tokenId)
    external override canTransfer(_tokenId) validNFToken(_tokenId) {
    require (idToOwner[_tokenId] == _from, NOT_OWNER);
    require (_to != address (0), ZERO_ADDRESS);
    address from = idToOwner[_tokenId];
    if (idToApproval[_tokenId] != address (0)) {
      delete idToApproval[_tokenId];
    }
    _removeNFToken(from, _tokenId);
    require (idToOwner[_tokenId] == _from, NOT_OWNER);
    ownerToNFTokenCount[_from] = ownerToNFTokenCount[_from] − 1;
    delete idToOwner[_tokenId];
    _addNFToken(_to, _tokenId);
    require (idToOwner[_tokenId] == address (0), NFT_ALREADY_EXISTS);
    idToOwner[_tokenId] = _to;
    ownerToNFTokenCount[_to] = ownerToNFTokenCount[_to].add(1);
}
```

```
contract NFToken {
    type Token is nonfungible asset nat
    type TokenApproval is nonfungible consumable asset nat
    balances : linking address <=> set Token
    approval : linking address <=> set TokenApproval
    ownerToOperators : linking address <=> {address}
    view canTransfer(_tokenId : nat) returns bool :=
        _tokenId in balances[msg.sender] or
        _tokenId in approval[msg.sender] or
        msg.sender in ownerToOperators[balances.ownerOf(_tokenId)]
    view validNFToken(_tokenId : nat) returns bool := balances.hasOwner(_tokenId)
    transaction transferFrom(_from : address, _to : address, _tokenId : nat):
        only when _to != 0x0 and canTransfer(_tokenId)
        if approval.hasOwner(_tokenId) {
            approval[approval.ownerOf(_tokenId)] −−[ _tokenId ]−> consume
        }
        balances[_from] −−[ _tokenId ]−> balances[_to]
}
```

Figure 4: A Solidity and a LANGUAGE-NAME implementation of the transferFrom function of the ERC-721 standard.

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

$$
\begin{array}{rcll}
\mathcal{Q} \ominus \texttt{empty} & = & \mathcal{Q} & \\
\texttt{empty} \ominus \mathcal{R} & = & \texttt{empty} & \\
\mathcal{Q} \ominus \texttt{every} & = & \texttt{empty} & \\
\texttt{every} \ominus \mathcal{R} & = & \texttt{every} & \textit{if } \mathcal{R} < \texttt{every} \\
\texttt{nonempty} - \mathcal{R} & = & \texttt{any} & \textit{if } \texttt{empty} < \mathcal{R} < \texttt{every} \\
! - \mathcal{R} & = & \texttt{empty} & \textit{if } ! \le \mathcal{R} \\
! - \texttt{any} & = & \texttt{any} & \\
\texttt{any} - \mathcal{R} & = & \texttt{any} & \textit{if } \texttt{empty} < \mathcal{R} < \texttt{every}
\end{array}
$$

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* Identifiers $\to$ Types $\cup \{\bot\}$ *as follows:*

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

We write $\textbf{dom}(\Gamma)$ to mean $\{x \in \text{Identifiers} \mid \Gamma(x) \neq \bot\}$, 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*

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

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

$$\mathcal{Q}\ T_\mathcal{Q} \le \mathcal{R}\ T_\mathcal{R} \iff T_\mathcal{Q} = T_\mathcal{R} \textit{ and } \mathcal{Q} \sqsubseteq \mathcal{R}$$

```
contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
        uint vote;
    }
    struct Proposal {
        bytes32 name;
        uint voteCount;
    }
    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal(proposalNames[i], ));
        }
    }
    function giveRightToVote(address voter) public {
        require(msg.sender == chairperson,
            "Only chairperson can give right to vote.");
        require(!voters[voter].voted,
            "The voter already voted.");
        voters[voter].weight = 1;
    }
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;
        proposals[proposal].voteCount += sender.weight;
    }
    function winningProposal() public view
            returns (uint winningProposal_) {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }
    function winnerName() public view
            returns (bytes32 winnerName_) {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

```
contract Ballot {
    type Voter is nonfungible asset address
    type ProposalName is nonfungible asset string
    chairperson : address
    voters : set Voter
    proposals : linking ProposalName <=> set Voter
    winningProposalName : string
    on create(proposalNames : set string):
        chairperson := msg.sender
        new Voter(chairperson) --> voters
        new ProposalName --[ proposalNames ]-> (\name. name <=> {}) --> proposals
    transaction giveRightToVote(voter : address):
        only when msg.sender = chairperson
        new Voter(voter) --> voters
    transaction vote(proposal : string):
        voters[msg.sender] --> proposals[proposal][msg.sender]
        if total proposals[proposal] > total proposals[winningProposalName] {
            winningProposalName := proposal
        }
    view winningProposal() returns string := winningProposalName
}
```

Figure 5: A Solidity and a LANGUAGE-NAME implementation of a simple voting contract.

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**

$$\begin{array}{ll}
C \in \textsc{ContractNames} & m \in \textsc{TransactionNames}\\
t \in \textsc{TypeNames} & x, y, z \in \textsc{Identifiers}\\
n \in \mathbb{Z} &
\end{array}$$

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

Figure 6: Abstract syntax of LANGUAGE-NAME.

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

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.

$$\frac{}{\Gamma \vdash \texttt{emptyval} : \texttt{empty}\ C\ \tau \dashv \Gamma}\ \textsc{Empty-Val}$$

$$\frac{\Gamma, x : \tau \vdash E : \sigma \dashv \Gamma}{\Gamma \vdash (\lambda x : \tau . E) : \texttt{every list !}\ (\tau \rightsquigarrow \sigma) \dashv \Gamma}\ \textsc{Transformer}$$

$$\frac{\Gamma \vdash x : \tau \dashv \Delta}{\Gamma \vdash \texttt{some}(x) : \texttt{! option}\ \tau \dashv \Delta}\ \textsc{Some}$$

$$\frac{\Gamma \vdash \overline{y : \tau} \dashv \Delta}{\Gamma \vdash \{\overline{x : \tau \mapsto y}\} \dashv \Delta}\ \textsc{Build-Rec}$$

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.

$$\frac{}{\Gamma, x : \tau \vdash x : \text{demote}(\tau) \dashv \Gamma, x : \tau}\ \textsc{Demote-Lookup}$$

$$\frac{}{\Gamma, x : \mathcal{Q}\ T \vdash x : \mathcal{Q}\ T \dashv \Gamma, x : \texttt{empty}\ T}\ \textsc{Lin-Lookup}$$

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

[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.

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

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.

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

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

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

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

[Add method typing as transformers]

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**

$$\frac{}{\Gamma, S : \tau \vdash S\ \texttt{provides}_{!}\ \tau}\ \textsc{Provide-One}$$

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

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

[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**.

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

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

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

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

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

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

$$\frac{}{\Gamma \vdash \texttt{everything selects}_{\texttt{every}}\ \tau}\ \textsc{Select-Everything}$$

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

$\boxed{\texttt{validSelect}(s, \mathcal{R}, Q)}$ We need to ensure that the resources to be selected are easily computable. In particular, we wish to enforce that we never select **everything** from a source containing **every** of something, nor do we use a selector like $\mathfrak{q}x : \tau$ s.t. E on a source contianing **every** of something. The following definition captures these restrictions.

$$\texttt{validSelect}(s, \mathcal{R}, Q) \iff \min(Q, \mathcal{R}) < \texttt{every} \text{ and } (Q = \texttt{every} \implies \exists \mathcal{L}.s = \mathcal{L})$$

$\boxed{\Gamma \vdash S\ \textbf{ok} \dashv \Delta}$ **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 $A \xrightarrow{s} (\lambda x : \tau . x) \longrightarrow B$. 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 $A \xrightarrow{s} f \longrightarrow B$. **[Actually, the type of f will probably be enough to distinguish the cases, but if we want to desgaur 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.

$$\frac{\Gamma \vdash A\ \texttt{provides}_{Q}\ \tau \qquad \Gamma \vdash s\ \texttt{selects}_{\mathcal{R}}\ \tau \qquad \texttt{validSelect}(s, \mathcal{R}, Q) \qquad \Delta = \texttt{update}(\Gamma, A, \Gamma(A) \ominus \mathcal{R}) \qquad \Delta \vdash f : \tau \rightsquigarrow \sigma \dashv \Delta \qquad \Delta \vdash B\ \texttt{accepts}\ \sigma}{\Gamma \vdash (A \xrightarrow{s} f \longrightarrow B)\ \textbf{ok} \dashv \texttt{update}(\Delta, B, \Delta(B) \oplus \min(Q, \mathcal{R}))}\ \textsc{Ok-Flo}$$

[TODO: Finish handling currying transformers.]

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

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

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

$$\frac{\Gamma \vdash E : \tau \dashv \Gamma \qquad \neg(\tau\ \texttt{asset})}{\Gamma \vdash \texttt{revert}(E)\ \textbf{ok} \dashv \Gamma}\ \textsc{Ok-Revert}$$

$$\frac{\Gamma \vdash E : \tau \dashv \Delta \qquad \neg(\tau\ \texttt{asset})}{\Gamma \vdash E\ \textbf{ok} \dashv \Delta}\ \textsc{Ok-Expr}$$

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

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

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

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

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

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

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

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.]**

$$\frac{Q \neq \texttt{every}}{\vdash_C (x : Q\ T)\ \textbf{ok}}\ \textsc{Ok-Field}$$

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. **[Need to ensure that nonfungible types are immutable.]**

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

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.

$$\frac{\texttt{fields}(C) = \overline{\texttt{this}.f : Q\ T} \qquad \texttt{this} : C, \overline{x : \tau}, \overline{\texttt{this}.f : \texttt{empty}\ T} \vdash S\ \textbf{ok} \dashv \Delta \qquad \forall}{\vdash_C (\texttt{on create}(\overline{x : \tau})\ \texttt{do}\ S)\ \textbf{ok}}$$

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

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

$\boxed{\texttt{Prog}\ \textbf{ok}}$ **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...]**

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

$\boxed{\texttt{decls}(C) = \overline{\texttt{Decl}}}$ **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)?]

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.]**