LANGUAGE-NAME: A DSL for Safe Blockchain Assets

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

[Authors/affiliations?]

1 INTRODUCTION

[Blockchain intro.]

Commonly proposed and implemented applications for the blockchain often revolve around the management of "digital assets." **[cite]** One of the most forms of these are a variety of smart contracts that managed assets called "tokens." There are many token standards, especially on the Ethereum blockchain **[cite]**, including **[ERC-20, ERC-721, ERC-777, ERC-1155]**, with others in various stages of the standardization process. Other applications or proposed applications for smart contracts include voting, supply chain management, auctions, lotteries, and other applications which require careful management of their respective assets **[cite/find examples]**.

LANGUAGE-NAME provides features to mark certain values as *assets* as well as a special construct called a *flow*, an abstraction representing an atomic transfer operation, which is widely applicable to smart contracts managing a variety of digital assets. These features combine to prevent common issues in smart contracts. Solidity **[cite]** and Vyper **[cite]**, the most common languages used to write smart contracts on the Ethereum blockchain **[cite]**, being general purpose languages, do not provide analgous support for managing assets.

Contributions. We make the following contributions with LANGUAGE-NAME.

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

2 LANGUAGE DESCRIPTION

A LANGUAGE-NAME program is made of many *contracts*, each containing *declarations*, such as *fields*, *types*, *transactions*, and *views*. A contract is the high-level grouping of functionality, just as in Solidity; each contract instance in LANGUAGE-NAME represents a contract on the blockchain. Fields function as the peristent storage of the contract, whose data is kept on the blockchain. Type declarations are the primary way to use the type system of LANGUAGE-NAME, providing a way of annotating values as assets (as well as other modifiers, discussed in Section 2.2). In place of Solidity's **function**, we have `transaction` and `view`. In LANGUAGE-NAME, a transaction can change the state of the contract, and a view cannot; it can, however, read the current state of the contract, just like the **view** modifier in Solidity. **[Explain how things map a little bit more. Also, this explanation basically is mostly useful if you already know Solidity. Probably should put more effort into explanation for people who don't know Solidity (at least, if we submit to a non-blockchain venue).]**

Figure 1 shows a simple contract declaring a type, a field, and a transaction, which implements the core functionality of ERC-20's `transfer` function (see Section **??** for more details on ERC-20).

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

Figure 1: A contract with a simple `transfer` function in LANGUAGE-NAME, which transfers `amount` tokens from the first to the second account. It is implemented with a single flow, which automatically checks all the preconditions to ensure the transfer is valid.

2.1 Syntax

Figure 2 shows a fragment of the syntax of the core calculus of LANGUAGE-NAME, which uses A-normal form 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???).]**

2.2 Modifiers

Modifiers can be used to place constraints on how values are managed: `asset`, `fungible`, `unique`, `immutable`, and `consumable`. A `asset` is a value that must not be reused or accidentally lost. A `fungible` value represents a quantity which can be **merged**, and it is **not** `unique`. A `unique` value can only exist in at most one storage; it must be `immutable`. A `immutable` value cannot be changed; in particular, it cannot be source or destination of a flow, the only state-changing construct in LANGUAGE-NAME**[this is the goal, anyway, which I think is true, but need to verify]**. A `consumable` value is an `asset` that it is sometimes appropriate to dispose of; however, this disposal must be done via the `consume` construct, a

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

Figure 2: A fragment of the abstract syntax of the core calculus of LANGUAGE-NAME, a simplified form of the surface language.

way of documenting that the disposal is intentional. All of these constraints, except for uniqueness, are enforced statically.

For example, ERC-20 tokens are `fungible`, while ERC-721 tokens are best modeled as being both `unique` and `immutable`. By default, neither is `consumable`, but one of the common extensions of both standards is to add a `burn` function, which allows tokens to be destroyed by users with the appropriate authentication. In this case, it would be appropriate to add the `consumable` modifier.

It supports data structures that make working with assets easier, such as *linkings*, a bidirectional map between keys and collections of values, with special operations to support modeling of "token accounts" (i.e., addresses which have a balance consisting of a set of tokens). A linking is essentially a `map K =>C V` with extra operations, where `K` is the key type, `V` is the value type, and `C` is some collection type constructor, such as `list` or `set`. These extra operations include querying which key owns which value, removing the need to maintain "parallel" data structures which separately keep track of various information about an asset; this operation is often useful (see Figure 4b for some examples). Though one can partially implemeny linkings as a library in Solidity, using the **unique** feature of LANGUAGE-NAME, we can guarantee that looking up the owner of a value which has a unique type is a **well-defined** operation.

2.3 Flows

LANGUAGE-NAME is built around the concept of a *flow*, an atomic, state-changing operation describing the transfer of a asset. Each flow has a *source* and a *destination*; they may optionally have a *selector* or a *transformer*, which default to **everything** and the identity transformer, respectively. The source of a flow *provides* values, the destination of the flow *accepts* these values, and the selector describes which subpart of the value(s) in the source should be transferred to the destination. 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 source, or if there is too much in the destination; for example, the latter may occur in the case of overflow. Flows can also fail for other reasons: a developer may specify that a certain flow must send all assets matching a predicate, but in addition specify an expected *quantity* that must be selected: any number, exactly one, or at least one.

[Discuss transformers]

2.4 Error Handling

Computation on blockchains like the Ethereum blockchain are grouped into units called *transactions*. **[which makes the use of the transaction keyword a little awkward, because multiple transaction calls can happen in a single transaction on the blockchain...]** Transactions either succeed or they fail and revert all changes. LANGUAGE-NAME also has transactional semantics: a sequence of flows will either all succeed, or, if a single flow fails, the rest will fail as well. If a sequence of flows fails, it "bubbles up" like an exception, until it either: a) reaches the top level, at which point the entire transaction fails; or b) reaches a `catch`, in which case only the changes made inside the corresponding `try` block will be reverted, and the code inside the `catch` block will be executed.

3 CASE STUDIES

3.1 ERC-20

ERC-20 **[cite]** is a standard for smart contracts that manage **fungible** tokens, and provides a bare-bones interface for this purpose. Each ERC-20 contract manages the "bank accounts" for its own tokens, keeping track of which users, identified by addresses, have some number of tokens. We focus on one core function from ERC-20, the `transfer` function. ERC-20 is one of the commonly implemented standards on the Ethereum blockchain **[cite]**. **[Cite all Solidity code]** Figure 3 shows a Solidity implementation of the ERC-20

```
contract EIP20 {
    mapping (address => uint256) private _balances;
    function transfer(address to, uint256 value)
        public returns (bool) {
        require(value <= _balances[msg.sender]);
        _balances[msg.sender] = _balances[msg.sender].sub(
            value);
        _balances[to] = _balances[to].add(value);
        return true;
    }
}
```

Figure 3: An implementation of ERC-20's `transfer` function in Solidity. All preconditions are checked manually. Note that we must include the `SafeMath` library (not shown), which checks for underflow/overflow, to use the `add` and `sub` functions.

function `transfer` (cf. Figure 1). Note that event code has been omitted, because LANGUAGE-NAME handles events in the same way

as Solidity. This example 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 automatic, descriptive error messages, such as "Cannot flow '<amount>' Token from account[<src>] to account[<dst>]: source only has <balance> Token.". 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 operations like increment and decrement or insert and delete.

3.2 ERC-721

ERC-721 **[cite]**, like ERC-20, is a token standard for the Ethereum blockchain. ERC-721 is a standard for tokens managing *nonfungible* tokens; that is, tokens that are unique. The ERC-721 standard requires many invariants hold, including: the tokens must be unique, the tokens must be owned by at most one account, at most one non-owning account can have "approval" for a token, and only if that token has been minted, 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 manage assets, it has features to help developers ensure that these correctness properties hold.

Figures 4a and 4b show implementations in Solidity and LANGUAGE-NAME, respectively, of the ERC-721 function transferFrom. The Solidity implementation is extracted from a reference implementation of ERC-721 given on the official Ethereum EIP page.

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 **unique**, we can be sure that we will not mint two of the same token. In addition to the invariants required by the specification, there are also internal invariants which the contract must maintain, such as the connection between idToOwner and ownerToNFTokenCount, which are handled automatically in the LANGUAGE-NAME version. This example demonstrates the benefits of having **unique** assets and the linking data structure built into the language itself.

3.3 Voting

One proposed use for the blockchain is smart contracts for managing voting **[cite]**. Figures 5a and 5b show the core of an implementation of a simple voting contract in Solidity and LANGUAGE-NAME, respectively. This example shows that LANGUAGE-NAME is suited for a range of applications, as we can use the **unique** and **immutable** modifiers to remove certain incorrect behaviors, shown in Figure 5.

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

3.4 The DAO attack

One of the most financially impactful bugs in a smart contract on the Ethereum blockchain was the bug in the DAO contract which allowed a large quantity of ether, worth about $50 million dollars at the time, to be stolen **[cite, verifying dollar amount]**. The bug was caused by a reentrancy-unsafe function in the contract, illustrated below. **[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, this attack could not have occurred for several reasons. Consider the following implementation of the same function in LANGUAGE-NAME given below.

```
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 performing the external call—without any developer intervention. Additionally, LANGUAGE-NAME forbids any reentrant call from an external source, a similar approach to the Obsidian language **[cite]**, which would also prevent more complicated reentrancy attacks.

4 DISCUSSION

5 RELATED WORK

[Obsidian, Scilla, Move, etc.?] [TODO: The safety guarantees provided by LANGUAGE-NAME differ from those provided by other languages because...Obsidian has similar concept of assets, but doesn't allow expressing immutability or uniqueness (could do fungibility via an interface, I suppose); this is probably going to be similar to other languages built around linear type systems. Obsidian's reentrancy scheme is slightly different, Nomos has a similar global lock, it seems.]

6 FUTURE WORK

7 CONCLUSION

A FORMALIZATION

A.1 Syntax

```
contract NFTokenf
  mapping(uint256 => address) idToOwner;
  mapping(uint256 => address) idToApproval;
  mapping(address => uint256) ownerToNFTokenCount;
  mapping(address => mapping(address => bool))
      ownerToOperators;

  modifier canTransfer(uint256 tokId) {
    address tokenOwner = idToOwner[tokId];
    require(tokenOwner == msg.sender||
            idToApproval[tokId] == msg.sender ||
            ownerToOperators[tokenOwner][msg.sender]);
    _;
  }
  function transferFrom(address _from, address _to, uint256
      tokId)
    external canTransfer(tokId) {
    require(idToOwner[tokId] != address(0));
    require(idToOwner[tokId] == _from);
    require(_to != address(0));
    if (idToApproval[tokId] != address(0)) {
      delete idToApproval[tokId];
    }
    require(idToOwner[tokId] == _from);
    ownerToNFTokenCount[_from] = ownerToNFTokenCount[_from]
        - 1;
    delete idToOwner[tokId];
    require(idToOwner[tokId] == address(0));
    idToOwner[tokId] = _to;
    ownerToNFTokenCount[_to] = ownerToNFTokenCount[_to].add(
        1);
  }
}
```

(a) A Solidity implementation of ERC-721's transferFrom.

```
contract NFTokenf
  type Token is unique immutable asset uint256
  type Approval is unique immutable consumable asset uint256
  balances : linking address ⇔ set Token
  approval : linking address ⇔ set Approval
  ownerToOperators : linking address ⇔ set address

  view canTransfer(tokId : uint256) returns bool :=
// `A in B` is true iff we can select `A` from `B`.
// It can be implemented efficiently if the LHS is hashable.
    tokId in balances[msg.sender] or
    tokId in approval[msg.sender] or
    msg.sender in ownerToOperators[balances.ownerOf(tokId)]

  transaction transferFrom(_from : address, _to : address, tokId
      : uint256):
    only when _to != 0x0 and canTransfer(tokId)
    if approval.hasOwner(tokId) {
      approval[approval.ownerOf(tokId)] --[ tokId ]-> consume
    }
    balances[_from] --[ tokId ]-> balances[_to]
}
```

(b) LANGUAGE-NAME implementation of ERC-721's transferFrom.

[We have public and private transactions...we could also have a public/private type? The difference being that public types can be transferred between contracts.] [Random note: We can optimize merge operations on unique types by dropping some checks.]

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 $\texttt{any}\ x : \tau$ s.t. $\texttt{if}\ b\ \texttt{then}\ x = y\ \texttt{else}\ false$]

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

A.2 Statics

Definition 1. *Define* $\boldsymbol{Quant} = \{\texttt{empty}, \texttt{any}, !, \texttt{nonempty}, \texttt{every}\}$, *and call any* $Q \in \boldsymbol{Quant}$ *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]

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

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

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

$Q \oplus \mathcal{R}$ 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.

```
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;


    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 (bytes32 winningProposal_) {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = proposals[p].name;
            }
        }
    }
}
```

(a) Solidity

```
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

    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
}
```

(b) LANGUAGE-NAME

Figure 5: A voting contract with a set of proposals, for which each user must first be given persmission to vote by the chairperson, assigned in the constructor of the contract (not shown)> Each user can vote exactly once for exactly one proposal. The proposal with the most votes wins.

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*

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

$C \in$ ContractNames $\quad m \in$ TransactionNames
$t \in$ TypeNames $\quad x, y, z \in$ Identifiers
$n \in \mathbb{Z}$

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

Figure 6: Abstract syntax of the core calculus of LANGUAGE-NAME.

Γ, Δ, Ξ ::= $\emptyset \mid \Gamma, x : \tau$ (type environments)

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

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

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

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

We write $\textbf{dom}(\Gamma)$ *to mean* $\{x \in$ 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*

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash x : \tau \dashv \Delta}{\Gamma \vdash \texttt{single}(x) : !\ C\ \tau \dashv \Delta}\ \textsc{Single}$$

$$\frac{\Gamma, x : \tau \vdash E : \sigma \dashv \Gamma, x : \pi \qquad \neg(\pi\ \texttt{asset})}{\Gamma \vdash (\lambda x : \tau.E) : !\ (\tau \rightsquigarrow \sigma) \dashv \Gamma}\ \textsc{Transformer}$$

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

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 **[probably should prove this to be sure]**.

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

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

$$\frac{f : \sigma \in \overline{y : \tau}}{\Gamma, x : \{\overline{y : \tau}\} \vdash x.f : \text{demote}(\sigma) \vdash \Gamma, x : \{\overline{y : \tau}\}}\ \textsc{Record-Demote-Lookup}$$

$$\frac{f : Q\ T \in \overline{y : \tau} \qquad \text{demote}(\sigma) \neq \sigma \qquad \overline{z : \sigma} = (\overline{y : \tau} \setminus \{f : Q\ T\}) \cup \{f : \texttt{empty}\ T\}}{\Gamma, x : \{\overline{y : \tau}\} \vdash x.f : Q\ T \vdash \Gamma, x : \{\overline{z : \sigma}\}}\ \textsc{Record-Lin-Lookup}$$

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.

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

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. **[Views should also probably check that we're packed? Or are we not concerned about it.]**

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

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

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

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

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

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

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

$\boxed{\texttt{elemtype}(T) = \sigma}$

$$\texttt{elemtype}(T) = \begin{cases} \sigma & \text{if } T = C\ \sigma \\ !\ T & \text{otherwise} \end{cases}$$

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

$$\frac{}{\Gamma, S : Q\ T \vdash S\ \texttt{provides}_Q\ \texttt{elemtype}(T)}\ \textsc{Provide-Var}$$

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

$\boxed{\Gamma \vdash \mathcal{D}\ \texttt{accepts}\ \tau}$ **Destination Typing** 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**.

$$\frac{}{\Gamma, D : Q\ T \vdash S\ \texttt{accepts}\ \texttt{elemtype}(T)}\ \textsc{Accept-Var}$$

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

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

$$\frac{\Gamma \vdash \mathcal{L} : Q\ T \dashv \Gamma}{\Gamma \vdash \mathcal{L}\ \mathbf{selects}_{Q}\ \mathbf{elemtype}(T)}\ \textsc{Select-Var}$$

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

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

$\boxed{\mathbf{validSelect}(s, \mathcal{R}, Q)}$ 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.

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

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

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.]**

$$\frac{\begin{array}{c}\Gamma \vdash A\ \mathbf{provides}_{Q}\ \tau \qquad \mathbf{immutable} \notin \mathbf{modifiers}(\Gamma(A)) \\ \Gamma \vdash s\ \mathbf{selects}_{\mathcal{R}}\ \tau \qquad \mathbf{validSelect}(s, \mathcal{R}, Q) \\ \Delta = \mathbf{update}(\Gamma, A, \Gamma(A) \ominus \mathcal{R}) \qquad \Delta \vdash f : \tau \rightsquigarrow \sigma \dashv \Delta \\ \Delta \vdash B\ \mathbf{accepts}\ \sigma \qquad \mathbf{immutable} \notin \mathbf{modifiers}(\Delta(B))\end{array}}{\Gamma \vdash (A \xrightarrow{s} f \longrightarrow B)\ \mathbf{ok} \dashv \mathbf{update}(\Delta, B, \Delta(B) \oplus \min(Q, \mathcal{R}))}\ \textsc{Ok-Flow}$$

[TODO: Finish handling currying transformers.]

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

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

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

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

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

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

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

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

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

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

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

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

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.]**

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

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.

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

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.

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

[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 pre-conditions 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.]**

Con ok **Contract Well-formedness**

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

Prog ok **Program Well-formedness**

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

Other Auxiliary Definitions. $\texttt{modifiers}(T) = \overline{M}$ **Type Modifiers**

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

$\text{demote}(\tau) = \sigma$ $\text{demote}_*(T_1) = T_2$ **Type Demotion** demote and demote$_*$ take a type and "strip" all the asset modifiers from it, as well as unfolding named type definitions. This process is useful, because it allows selecting asset types without actually having a value of the desired asset type. Note that demoting a transformer type changes nothing. This is because a transformer is **never** an asset, regardless of the types that it operators on, because it has no storage.

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

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

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

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

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

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

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

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

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