Psamathe: A DSL for Safe Blockchain Assets

ACM Reference Format:
. 2020. Psamathe: A DSL for Safe Blockchain Assets. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

[Authors/affiliations?]

1 INTRODUCTION

Blockchains are becoming more commonly used as platforms for applications, called *smart contracts*, which automatically manage transactions in an unbiased, mutually agreed-upon way. 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 [6], including ERC-20, ERC-721, ERC-777, ERC-1155 [2] [3] [4] [1], 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]**.

However, smart contracts must be carefully reviewed before being deployed, as they cannot be patched if a security vulnerability is discovered. Despite this extra care, discoveries of vulnerabilities still occur regularly, often costing large amounts of money. **[Some examples]**

Psamathe provides features to mark certain values as *assets*, with various *modifiers* to control their use, as well as a new abstraction, called a *flow*, representing an atomic transfer operation, which is widely applicable to smart contracts managing digital assets. These features combine in a novel way 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 Psamathe.

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

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

2 LANGUAGE

A Psamathe program is made of *contracts*, each containing *declarations*, such as *fields*, *types*, and *functions*. A contract is the high-level grouping of functionality, just as in Solidity; each contract instance in Psamathe 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 Psamathe, providing a way of annotating values as assets (as well as other modifiers, discussed in Section 2.3). We distinguish between three types of function: *transactions*, *views*, and *transformers*. Transactions and views are both kinds of *method*, which may access the fields of a contract; a transaction can read **and** write fields, whereas a view may only read fields. A transformer is a **pure** function: it may contain flows and other statements, but it cannot mutate the state of any contract.

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).

```
1 contract ERC20 {
2   type Token is fungible asset uint256
3   balances : map address => Token
4   transaction transfer(dst : address, amt : uint256):
5     balances[msg.sender] −−[ amt ]−> balances[dst]
6 }
```

Figure 1: A contract with a simple `transfer` function in Psamathe, 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 Psamathe, which uses A-normal form and makes several other simplifications to the surface Psamathe language. These simplifications are performed automatically by the compiler. **[TODO: We have formalized this core calculus (in K???).]**

2.2 Flows

Psamathe 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

$\mathfrak{q}$	::= ! | **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 Psamathe, a simplified form of the surface language.

transferred to the destination. Selectors act like filters from functional programming, or like the WHERE clause of a SELECT in a SQL database. Using a transformer in a flow is like mapping a function over the collection of values being flowed.

Using flows provides the following advantages over the typical approach **[come up with a clearer name for this]**:

- **Precondition checking**: For a flow to succeed, the source must have enough assets and the destination must be capable of receiving the assets flowed. For example, a flow of money would fail if there is not enough in the source, or if there is too much in the destination; the latter may occur because 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. The former will always succeed, but the latter two may cause the flow to fail.
- **Data-flow tracking**: It is clear where the resources are flowing from the code itself, which may not be apparent in complicated contracts, 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 assets, such as ether from being consumed.
- **Error messages**: When a flow fails, Psamathe provides automatic, descriptive error messages, such as "Cannot flow '<amount>' Token from account[<src>] to account[<dst>]: source only has <balance> Token.". By default, Solidity provides no error messages, 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.

2.3 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 the source or destination of a flow, the only state-changing construct in Psamathe**[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 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.

Psamathe also 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 an *account* holding a collection of assets. A `linking K <=> C V` can be thought of as 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`. Specifically, a linking implements the following interface for any collection type `C`. Note that the specific syntax is not allowed in an interface declaration, but is used because it is syntax used in the language to write the linking type. **[Either need to finish up the interface proposal, or write this in another way]**

```
interface linking K <=> C V where C is collection {
  empty : linking K <=> C V
  get : (linking K <=> C V) ~> K ~> C V
  put : (linking K <=> C V) ~> K ~> V ~> linking K <=> C V
  hasOwner : (linking K <=> C V) ~> V ~> bool
  ownerOf : (linking K <=> C V) ~> V ~> K
}
```

Note that some operations may result in errors. See Figure 4b for some example uses of this structure. Note that, though it is possible to partially implement linkings as a library in Solidity, in Psamathe, we can guarantee that looking up the owner of a value which has a `unique` type is a **well-defined** operation.

[Discuss transformers]

2.4 Error Handling

Computation on blockchains like the Ethereum blockchain is grouped into units called *transactions*. Transactions either succeed or they fail and revert all changes. Psamathe 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.

2.5 Comparison with Solidity

Contracts in Psamathe map directly to contracts in Solidity. Transactions, views, and transformers all map to the concept of **function** in Solidity: a transaction is a function that is public by default, a view is a function with the **view** modifier, and a transformer is a function with the **pure** modifier. **[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).]**

Before version 0.6.0, Solidity did not have a try-catch construct. The resulting try-catch construct added behaves similarly to that of Psamathe, as both will revert the changes made when an exception occurs. However, Solidity's try-catch is structed in an atypical way, only allowing catching errors from a single expression, which must be an external call or a contract creation call. **[cite]**

3 EXAMPLES

[section intro]

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 function transfer (cf. Figure 1). Note that event code has been omitted, because Psamathe handles events in the same way as Solidity. This example shows the advantages of the flow abstraction in precondition checking and error messages. **[rephrase, because you can't see the error messages, but it's still part of the advantages...]**

```
contract ERC20 {
  mapping (address => uint256) balances;
  function transfer(address dst, uint256 amt)
    public returns (bool) {
    require(amt <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender].sub(amt);
    balances[dst] = balances[dst].add(amt);
    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.

In this case, the balance of the sender must be at least as large as 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.

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 Psamathe 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 Psamathe, 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 Psamathe 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 Psamathe 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 Psamathe, respectively. This example shows that Psamathe is suited for a range of applications, as we can use the **unique** modifier to remove certain incorrect behaviors, shown in Figure 5. In this application, the use of **unique** ensures that each user, represented by an *address*, can be given permission to vote at most once, while the use of **asset** ensures that votes are not lost or double-counted.

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 Solidity function shown below allows a user to withdraw their balance of ether whenever they wish. It does this by first transferring the ether from the contract to the user's address, then decreasing the user's balance. However, in Ethereum, a transfer of ether also calls a special function on the receiving address, when it is a contract address, called a *fallback* function. In this

```
contract NFToken {
  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 NFToken {
  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) Psamathe implementation of ERC-721's transferFrom.

function, arbitrary code may be executed, and because **call**.value() was used, this call has access to all of the remaining gas of the transaction. A malicious contract can use this to perform a *reentrant* call back into the withdrawBalance function, and withdraw their balance again—the user's balance has not been decreased yet. The fix in this case is a simple swap of the last two lines of the function, but reentrancy issues can be more complicated than this. **[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 Psamathe, this attack could not have occurred for several reasons. Consider the following implementation of the same function in Psamathe 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, Psamathe 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 Psamathe differ from those provided by other languages because...Obsidian has similar concept of assets, but doesn't allow expressing immutability, uniqueness, or consumability (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 CONCLUSION AND FUTURE WORK

[Conclusion]

```
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;
  }
}
```

(a) Solidity implementation of the core voting functions [5].

```
contract Ballot {
  type Voter is unique immutable asset address
  type ProposalName is unique immutable asset string

  chairperson : address
  voters : set Voter
  proposals : linking ProposalName <=> set Voter

  transaction giveRightToVote(voter : address):
    only when msg.sender = chairperson
    new Voter(voter) −−> voters


  transaction vote(proposal : string):
    voters −−[ msg.sender ]−> proposals[proposal]
}
```

(b) Psamathe implementation of the core voting functions.

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.

In the future, we plan to fully implement the Psamathe language, including proofs of all of its safety properties. We also hope to perform additional studies on the benefits of the language including conducting longer case studies and comparing the efficiency of the resulting programs. We would also like to perform a user study to evaluate the usability of the flow abstraction and the design of the language itself.

REFERENCES

[1] [n.d.]. EIP 1155: ERC-1155 Multi Token Standard. https://eips.ethereum.org/EIPS/eip-1155
[2] [n.d.]. EIP 20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20
[3] [n.d.]. EIP 721: ERC-721 Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721
[4] [n.d.]. EIP 777: ERC777 Token Standard. https://eips.ethereum.org/EIPS/eip-777
[5] [n.d.]. Solidity by Example. https://solidity.readthedocs.io/en/v0.7.0/solidity-by-example.html Accessed: 2020-07-28.
[6] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

A FORMALIZATION

A.1 Syntax

[Sets can only contain immuatble types?] [Note that `unique` is not allowed if the type can contain a transformer.]

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

τ `asset` **Asset Types**

$C \in$ 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)
$\mathcal{L}, \mathcal{M}$	::=	x | $\mathcal{L}.x$ | $\mathcal{L}[\mathcal{L}]$	(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 Psamathe.

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

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

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

τ **consumable** Consumable Types

$(\mathcal{Q}\ T)$ **consumable** $\Leftrightarrow$**consumable** $\in$ **modifiers**(T) or
$\neg((\mathcal{Q}\ T)$ **asset**) or
$(T = C\ \tau$ and τ **consumable**) or
$(T = \{\overline{y : \sigma}\}$ and $\forall x : \tau \in \overline{y : \sigma}.(\sigma$ **consumable**$))$

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

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

$$
\begin{array}{rcll}
\mathcal{Q} \oplus \textbf{empty} & = & \mathcal{Q} & \\
\mathcal{Q} \oplus \textbf{every} & = & \textbf{every} & \\
\textbf{nonempty} \oplus \mathcal{R} & = & \textbf{nonempty} & \textit{if } \textbf{empty} < \mathcal{R} < \textbf{every} \\
! \oplus \mathcal{R} & = & \textbf{nonempty} & \textit{if } \textbf{empty} < \mathcal{R} < \textbf{every} \\
\textbf{any} \oplus \textbf{any} & = & \textbf{any} &
\end{array}
$$

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

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

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

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

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

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

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

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

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. **[TODO: Single rule should work for any fungible (or mergeable??) type]**

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

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

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

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

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]**.

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

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

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

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

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.

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

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.

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

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

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

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

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.

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

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

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

$$\textbf{elemtype}(T) = \begin{cases} \textbf{elemtype}(S) & \text{if } \textbf{type}\ T\ \textbf{is}\ \overline{M}\ S \\ \sigma & \text{if } T = C\ \sigma \\ !\ T & \text{otherwise} \end{cases}$$

$\boxed{\Gamma \vdash \mathcal{L} : \tau}$ **Non-consuming location typing** This judgement is exclusively used for checking the types of the storages in a flow. It is safe to use because the flow rule handles updating the storages after the flow. **[Would be nice to combine the linking and map rules, and especially nice to avoid special casing them altogether. Could be done with a "keytype(T)" function. Maybe we could make it work on sets of `mapitem` or `link`, and then avoid even defining maps and linkings.]**

$$\frac{(x\ \textbf{immutable}) \notin \Gamma}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Loc-Var}$$

$$\frac{\Gamma \vdash \mathcal{L} : Q\ T \qquad (f : \sigma) \in \textbf{fields}(T) \qquad \neg(T\ \textbf{immutable})}{\Gamma \vdash \mathcal{L}.f : \sigma}\ \textsc{Loc-Member}$$

$$\frac{\Gamma \vdash \mathcal{L} : Q\ (\textbf{map}\ \tau\ \Rightarrow\ \sigma) \qquad \Gamma \vdash \mathcal{M} : \tau}{\Gamma \vdash \mathcal{L}[\mathcal{M}] : \sigma}\ \textsc{Loc-Map}$$

$$\frac{\Gamma \vdash \mathcal{L} : Q\ (\textbf{linking}\ \tau\ \Leftrightarrow\ \sigma) \qquad \Gamma \vdash \mathcal{M} : \tau}{\Gamma \vdash \mathcal{L}[\mathcal{M}] : \sigma}\ \textsc{Loc-Linking}$$

[How to handle immutable properly?] $\boxed{\Gamma \vdash \mathcal{S}\ \textbf{provides}_Q\ \tau}$ **Source Typing**

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

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

$\boxed{\Gamma \vdash \mathcal{D}\ \textbf{accepts}\ \tau}$ **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**.

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

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

$\boxed{\Gamma \vdash s\ \textbf{selects}_Q\ \tau}$ **Selectors**

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

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

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

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

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

$\boxed{\Gamma \vdash S\ \textbf{ok} \dashv \Delta}$ **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.]**

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

[TODO: Finish handling currying transformers.]

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

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

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

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

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

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

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

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

$\vdash_C$ `Decl` ok **Declaration Well-formedness**

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

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

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

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 when it's modifiers are all compatible with each other, and it has the **asset** modifier if its base type is an asset. The latter restriction isn't strictly necessary, but is intended to help developers realize which types are assets without unfolding the entire type definition. **[TODO: What if the base type is a non-consumable asset, and we write consumable. Is that okay? Probably not.]**

$$\frac{\begin{array}{c}T\ \mathbf{asset} \Rightarrow \mathbf{asset} \in \overline{M} \\ \mathbf{unique} \in \overline{M} \Rightarrow \mathbf{immutable} \in \overline{M} \\ \mathbf{fungible} \in \overline{M} \Rightarrow \mathbf{unique} \notin \overline{M}\end{array}}{\vdash_C (\mathbf{type}\ t\ \mathbf{is}\ \overline{M}\ T)\ \mathbf{ok}}\ \textsc{Ok-Type}$$

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

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

`Con` ok **Contract Well-formedness**

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

`Prog` ok **Program Well-formedness**

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

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

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

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

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

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

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

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

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

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

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

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

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

A.3 WIP ideas

Definition 5. *Let* $\textbf{mgu}(\tau, \sigma)$ *denote the most general unifier of* τ *and* σ *such that if* $\textbf{mgu}(\tau, \sigma) = \theta$, *then* $\theta(\tau) = \sigma$; *if there is no unifier, then* $\textbf{mgu}(\tau, \sigma) = \bot$.

$$\frac{\emptyset \vdash S\ \textbf{ok} \dashv \Gamma \qquad \Gamma \vdash E : \tau \dashv \Delta \qquad \forall y. \neg(\Delta(y)\ \textbf{asset})}{(x : \tau := S\ \textbf{return}\ E)\ \textbf{ok}}\ \textsc{Init-Ok}$$

$$\frac{\begin{array}{c}\textbf{interface}\ I(\pi)\ \textbf{where}\ \overline{t\ \textbf{is}\ \overline{M}}\ \{\overline{\textbf{type}\ s\ \overline{x : \tau}}\} \\ \textbf{implementation}\ x\ \textbf{of}\ I(\sigma)\ \{\ldots\} \\ \textbf{mgu}(\pi, \sigma) \circ \textbf{mgu}(\sigma, \tau) = \theta \qquad \theta \neq \bot \\ \forall i. \forall N \in \overline{M_i}. \theta^{-1}(t_i)\ \textbf{is}\ \theta(N)\end{array}}{I(\tau)\ \textbf{resolvable}_x}\ \textsc{Resolvable-Impl}$$

$$\frac{\begin{array}{c}\overline{t} \subseteq \textbf{vars}(\tau) \\ \forall z : \pi \in \overline{y : \sigma}. \textbf{vars}(\pi) \subseteq \textbf{vars}(\tau) \cup \overline{s}\end{array}}{(\textbf{interface}\ I(\tau)\ \textbf{where}\ \overline{t\ \textbf{is}\ \overline{M}}\ \{\overline{\textbf{type}\ s\ \overline{y : \sigma}}\})\ \textbf{ok}}\ \textsc{Interface-Ok}$$

$$\frac{\begin{array}{c}\textbf{interface}\ I(\tau)\ \textbf{where}\ \overline{t\ \textbf{is}\ \overline{M}}\ \{\overline{\textbf{type}\ s\ \overline{x : \tau}}\} \\ \overline{x : \pi} = \overline{x : \tau[\overline{\tau}/\overline{t}][\overline{\sigma}/\overline{s}]} \\ \forall F \in \overline{x : \rho := S\ \textbf{return}\ E}. F\ \textbf{ok}\end{array}}{(\textbf{implementation}\ x\ \textbf{of}\ I(\sigma)\ \{\overline{\textbf{type}\ s\ \textbf{is}\ \pi}\ \overline{x : \rho := S\ \textbf{return}\ E}\})\ \textbf{ok}}\ \textsc{Impl-Ok}$$

[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: We can implement preconditions (i.e., "only when") with just flows by doing something like:

```
1 { 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.