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, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Blockchains are increasingly used as platforms for applications called *smart contracts*, which automatically manage transactions in an unbiased, mutually agreed-upon way. Commonly proposed and implemented applications often manage *digital assets*, such as smart contracts for supply chain management [15], healthcare [14], and other applications which require careful management of their respective assets such as voting, crowdfunding, or auctions [12]. One of the most common kinds of contract is a *token contract*—about 73% of high-activity contracts are token contracts [17]. On the Ethereum blockchain [21], there are many token standards, including ERC-20, ERC-721, ERC-777, ERC-1155 [4–7].

Smart contracts cannot be patched after being deployed, even if a security vulnerability is discovered. Developers must carefully review contracts, and some use an independent auditing service to help with this process. Despite this extra care, discoveries of vulnerabilities still occur regularly, often costing large amounts of money. The well-known DAO attack [20], discussed in Section 3.4, caused the loss of over 40 million dollars, and was due to a reentrancy issue. Some estimates suggest that as much as 46% of smart contracts may have some vulnerability [16].

Psamathe (/sɑmɑθi/) is a new programming language we are designing around a new abstraction, a *flow*, representing an atomic transfer operation, which is useful in smart contracts managing digital assets. Flows allow the encoding of semantic information about the flow of assets into the code. The Psamathe language will also provide features to mark types as *assets*, with various *modifiers* to control their use, which combine with flows to make some classes of bugs impossible. Solidity, the most commonly-used language for writing smart contracts on the Ethereum blockchain [8], does not make any effort to provide analogous support for managing assets. Additionally, typical smart contracts are more **concise** in Psamathe because it handles common patterns and pitfalls automatically.

2 LANGUAGE

A Psamathe program is made of *contracts*, each containing *declarations*: *fields*, *types*, and *functions*. A contract is the high-level grouping of functionality; each contract instance in Psamathe represents a contract on the blockchain. Fields function as the persistent 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 types as assets (as well as other modifiers, discussed in Section 2.4). We distinguish between three types of function: *transactions*, *views*, and *transformers*. Transactions and views are both *methods*, 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, which is not part of any contract: it may contain flows and other statements, and it may mutate **local** state, but it cannot mutate the state of any contract.

Figure 1 shows a simple contract declaring a contract, a type, a field, and a transaction, which implements the core functionality of ERC-20's transfer function; see Section 3.1 for more details on ERC-20.

```
contract ERC20 {
 type Token is fungible asset uint256
 balances : map address => Token
 transaction transfer(dst : address, amt : uint256):
  balances[msg.sender] --[ amt ]-> balances[dst]
}
```

Figure 1: A contract with a simple transfer function in Psamathe, which transfers amount tokens from the sender's account to the destination 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 values. 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. The source and destination of a flow are *storages*, which are either variables or one of the special storages **`new`** *t* or **`consume`**. A selector is a function that takes as input the current value of the storage, and outputs the value(s) to select. Selectors act can be used like a filter in functional programming, or

Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

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

like a WHERE in SQL. Using a transformer in a flow is like mapping a function over the collection of values being flowed.

The example in Figure 1 contains a single flow, on line 5; the source is `balances[msg.sender]`, the selector is `amt`, and the destination is `balances[dst]`. Because the type of assets being flowed is `Token`, defined as a `fungible asset uint256`, the actual computation will use the `uint256` type. When the transaction is executed, it will (1) check that `balances[msg.sender] >= amt`, (2) check that adding `amt` to `balances[dst]` will not overflow it, (3) subtract `amt` from `balances[msg.sender]`, (4) and add `amt` to `balances[dst]`.

Using a *flow-based* approach provides the following advantages over the typical *assignment-based* approach most languages use (e.g., incrementing, then decrementing):

- **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 creating such messages by encoding the necessary semantic information into the program, instead of using low-level operations like increment and decrement or insert and delete.

2.3 Type Quantities

When a Psamathe program is type checked, each variable has an associated *type quantity* at every point in the program, which provides an approximation of the number of values the variable holds. The list of allowed type quantities is: **empty**, **any**, !, **nonempty**, **every** (note "!" means "exactly one"). Only **empty** asset variables may be dropped, ensuring that unused assets not dropped. Type quantities are **not** required in the surface language, and they will be added automatically; for example, `var toks : set Token := {}` is inferred to be an `empty set Token`, whereas `var C : Bank := new Bank()` is inferred to be `! Bank`.

Flows are the primary method of updating type quantities. For example, after checking the flow `A --[ everything ]-> B`, it must be that `A` is empty, and depending on the type quantity that `A` used to have, it may now be that `B` is also empty (if `A` was empty), nonempty, or any other type quantity (other than **every**). Type quantities can also be updated by expressions, which can move values to other variables. For example, after checking `var x : (Token, Token) := (a, b)`, creating a new pair, both `a` and `b` are **empty**, as the assets they used to hold are now in `x`.

The type quantity system provides the benefits of *linear types*, but gives a more precise analysis of the flow of values in a program. Many variables are supposed to hold exactly one value; corresponding to being non-null in languages with nulls, like Java. The type quantity system allow temporarily "emptying out" a variable, while statically ensuring that it contains exactly one value before the transaction exits. Type quantities can also specify more precise types for variables, such as declaring an argument to a transaction must be a `nonempty list`. Finally, type quantities can specify **expectations** about program behavior in a way that can be automatically checked. For example, suppose that `A : set address` and `B : empty address`, and consider the following flow

```
A --[ ! addr such that isAdmin(addr) ]-> B
```

It must now be that `B` is `! address` if the flow was successful, and we have made it clear to the program and the reader that there should be exactly one admin in `A`.

2.4 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` and an `asset` to ensure it is not duplicated. 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. 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. Uniqueness is enforced at creation time by dynamically checking that the new value has never been created before. After this initial check, we can be sure that the values are unique because the type system will not allow an existing unique value to be duplicated. **[What if someone passes us an argument of a unique type in a public transaction? Is that even possible?]**

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 to support modeling of an *account* holding a collection of assets. A `linking K <=> C V` is a `map K => C V` with additional operations, where `K` is the key type, `V` is the value type, and `C` is some collection type constructor, such as `list` or `set`. The additional operations supported are `L.hasOwner(v)` and `L.ownerOf(v)`, where `L : linking K <=> C V` and `v : V`. `L.hasOwner(v)` returns true if and only if there is some `k : K` such that `v in L[k]` is true. `L.ownerOf(v)` returns some `k : K` such that `v in L[k]`, if it exists. If it does not, it throws an error. **[I have some ideas about adding interfaces to the language, which I would love to do but it would complicate things.]** See Figure 3b for some example uses of this structure. 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 in linking, if it has a `unique` type, is a **well-defined** operation.

2.5 Error Handling

Computation on blockchains like the Ethereum blockchain is grouped into units called *transactions*. Transactions in the same *block* are executed in a sequential order, but may be nondeterministically ordered within the block, at the discretion of the miner. 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.

2.6 Comparison with Solidity

The concept of contracts in Psamathe maps directly Solidity's concept of contracts. 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. A type declaration in Psamathe is best modeled by a **struct** in Solidity, although type declarations are more flexible: Solidity has does not support any of the modifiers that Psamathe does (note Solidity has its own concept of modifiers that is unrelated).

The Psamathe language provides all of the primitive types that Solidity does, with additional support for types such as linkings, explained above. Solidity also exhibits several atypical behaviors, such as acting as those all keys in a mapping exist, and providing the default value whenever a new key is used. This leads to ad-hoc workarounds, like those shown in Sections 3.2 and 3.3.

Before version 0.6.0, for the first four years since its release five years ago, Solidity did not have a try-catch construct [13]. The newly-added try-catch construct behaves similarly to that of Psamathe, as both will revert the changes made when an exception occurs. However, Solidity's try-catch is structured in an atypical way, only allowing catching errors from a single expression, which must be an external call or a contract creation call, whereas Psamathe behaves in a more standard way, allowing catching all errors from the try-block.

3 EXAMPLES

In this section, we show several example smart contracts, with a Solidity implementation and a Psamathe implementation, and discuss the benefits of each implementation. In each example, we only show the fragment of the code that is necessary for space reasons, but the full Solidity and Psamathe implementations can be found in our repository [1]. **[Do this]**

3.1 ERC-20

ERC-20 is a standard for smart contracts that manage **fungible** tokens, and provides a bare-bones interface for this purpose. ERC-20 is one of the commonly implemented standards on the Ethereum blockchain. 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. Figure 5 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 flows in precondition checking, data-flow tracking, and error messages. 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, 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

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

 function transferFrom(address src, address dst, uint256 tokId)
        external {
  require(idToOwner[tokId] == msg.sender ||
        idToApproval[tokId] == msg.sender ||
        ownerToOperators[idToOwner[tokId]][msg.sender]);
  require(idToOwner[tokId] != address(0));
  require(idToOwner[tokId] == src && dst != address(0));
  if (idToApproval[tokId] != address(0)) {
   delete idToApproval[tokId];
  }
  ownerToNFTokenCount[src] = ownerToNFTokenCount[src] − 1;
  idToOwner[tokId] = dst;
  ownerToNFTokenCount[dst] = ownerToNFTokenCount[dst].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

 transaction transferFrom(src : address, dst : address, tokId : uint256):
  // `A in B` is true iff we can select `A` from `B`.
  // It can be implemented efficiently if the LHS is hashable.
  only when tokId in balances[msg.sender] or
         tokId in approval[msg.sender] or
         msg.sender in ownerToOperators[balances.ownerOf(tokId)]
  if approval.hasOwner(tokId) {
   approval[approval.ownerOf(tokId)] −−[ tokId ]−> consume
  }
  balances[src] −−[ tokId ]−> balances[dst]
}
```

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

Figure 3: Implementation of ERC-721's transferFrom function, which sends a specific token from the src account to the dst account. It also must clear the approval of the token, if any.

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 3a and 3b 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. Note that all helper functions and modifiers have been unfolded, and some redundant code, present in the original implementation, has been removed for space reasons.

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 having the linking data structure built into the language.

3.3 Voting

One proposed use for the blockchain is smart contracts for managing voting [12]. Figures 4a and 4b 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 4. 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. Additionally, the Solidity implementation is more verbose than the Psamathe implementation because it must work around the limitations of Solidity's mappings. In this example, the declaration of the weight and voted members of the Voter struct exist so that the contract can tell whether a voter has the default values, was authorized to vote, or has already voted.

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 over 40 million dollars at the time, to be stolen [17]. 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 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

```
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 [2].

```
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 4: A voting contract with a set of proposals, for which each user must first be given permission 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.

```
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 5: An implementation of ERC-20's transfer function in Solidity from one of the reference implementations [3]. 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.

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, we expect that our compiler should be able 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 [10], which would also prevent more complicated reentrancy attacks.

4 DISCUSSION

5 RELATED WORK

There have been many new blockchain languages proposed, such as Flint [18], Move [9], Nomos [11], Obsidian [10], and Scilla [19].

Scilla and Move were proposed as intermediate-level languages, whereas Psamathe is a high-level language.

Obsidian, Move, Nomos, and Flint use linear types to manage assets, similarly to how Psamathe uses type quantities. None of the these languages have flows, which encode the semantics of the transfer into the code itself. **[Not really satisfied with this explanation...] [desribe why flows are a good alternative to whatever mechanisms these languages provide]. [I would like to make an argument that flows are more intuitive than ownership, but can't really do that.]** Additionally, none of these languages provide support for all the modifiers that Psamathe does. Type quantities provide a more precise analysis of the behavior of a program than linear types.

Obsidian and Nomos provide reentrancy protection via locking, as Psamathe does. Scilla requires that all external calls occur at the end of a transition, but this restriction makes it impossible to implement some ERC standards. Move requires that the module dependency graph be acyclic, which is unenforceable on Ethereum, our target blockchain.

6 CONCLUSION AND FUTURE WORK

We have presented the Psamathe langauge for writing safer smart contracts. Psamathe uses the new flow abstraction, assets, and modifiers to provide safety guarantees for smart contracts. We showed several examples of smart contracts in both Solidity and Psamathe, showing that Psamathe is capable of expressing common smart contract functionality in a concise manner, while retaining key safety properties.

In the future, we plan to fully implement the Psamathe language, and complete proofs of all of its safety properties. We also hope to perform additional studies on the benefits of the language including conducting case studies, comparing the efficiency of the resulting programs, and exploring the applicability of flows to other domains where careful asset management is important, outside of the blockchain. Finally, we would also like to perform a user study to evaluate the usability of the flow abstraction and the design of the language itself, as well as comparing it to Solidity.

REFERENCES

[1] [n.d.]. Psamathe. https://github.com/ReedOei/Psamathe
[2] [n.d.]. Solidity by Example. Retrieved 2020-07-28 from https://solidity.readthedocs.io/en/v0.7.0/solidity-by-example.html
[3] [n.d.]. Tokens. Retrieved 2020-08-03 from https://github.com/ConsenSys/Tokens
[4] 2015. EIP 20: ERC-20 Token Standard. Retrieved 2020-07-28 from https://eips.ethereum.org/EIPS/eip-20
[5] 2017. EIP 777: ERC777 Token Standard. Retrieved 2020-07-28 from https://eips.ethereum.org/EIPS/eip-777
[6] 2018. EIP 1155: ERC-1155 Multi Token Standard. Retrieved 2020-07-28 from https://eips.ethereum.org/EIPS/eip-1155
[7] 2018. EIP 721: ERC-721 Non-Fungible Token Standard. Retrieved 2020-07-28 from https://eips.ethereum.org/EIPS/eip-721
[8] 2020. Ethereum for Developers. Retrieved 2020-07-31 from https://ethereum.org/en/developers/
[9] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources.
[10] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. arXiv:cs.PL/1909.03523
[11] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2019. Resource-aware session types for digital contracts. *arXiv preprint arXiv:1902.06056* (2019).
[12] Chris Elsden, Arthi Manohar, Jo Briggs, Mike Harding, Chris Speed, and John Vines. 2018. Making Sense of Blockchain Applications: A Typology for HCI. In *CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. 1–14. https://doi.org/10.1145/3173574.3174032
[13] Elena Gesheva. 2020. Solidity 0.6.x features: try/catch statement. Retrieved 2020-07-29 from https://blog.ethereum.org/2020/01/29/solidity-0.6-try-catch/
[14] Harvard Business Review. 2017. The Potential for Blockchain to Transform Electronic Health Records. Retrieved February 18, 2020 from https://hbr.org/2017/03/the-potential-for-blockchain-to-transform-electronic-health-records
[15] IBM. 2019. Blockchain for supply chain. Retrieved March 31, 2019 from https://www.ibm.com/blockchain/supply-chain/
[16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309
[17] Gustavo Oliva, Ahmed E. Hassan, and Zhen Jiang. 2019. An Exploratory Study of Smart Contracts in the Ethereum Blockchain Platform. *Empirical Software Engineering* (11 2019). https://doi.org/10.1007/s10664-019-09796-5
[18] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. 218–219.
[19] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687* (2018).
[20] Emin Gün Sirer. 2016. Thoughts on The DAO Hack. Retrieved July 29, 2020 from http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/
[21] 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 immutable 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* $\mathbf{Quant} = \{\mathtt{empty}, \mathtt{any}, !, \mathtt{nonempty}, \mathtt{every}\}$, *and call any* $Q \in \mathbf{Quant}$ *a* type quantity. *Define* $\mathtt{empty} < \mathtt{any} < ! < \mathtt{nonempty} < \mathtt{every}$.

$\boxed{\tau\ \mathtt{asset}}$ Asset Types

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

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

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

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

Figure 6: Abstract syntax of the core calculus of Psamathe.

$$\Gamma, \Delta, \Xi \quad ::= \quad \emptyset \mid \Gamma, x : \tau \mid \Gamma, x\ \mathtt{immutable} \quad \text{(type environments)} \qquad \textit{such that}$$

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

$$\begin{aligned}
(\mathcal{Q}\ T)\ \mathtt{consumable} \Leftrightarrow & \mathtt{consumable} \in \mathtt{modifiers}(T) \text{ or} \\
& \neg((\mathcal{Q}\ T)\ \mathtt{asset}) \text{ or} \\
& (T = C\ \tau \text{ and } \tau\ \mathtt{consumable}) \text{ or} \\
& (T = \{\overline{y : \sigma}\} \text{ and } \forall x : \tau \in \overline{y : \sigma}.(\sigma\ \mathtt{consumable}))
\end{aligned}$$

$\mathcal{Q} \oplus \mathcal{R}$ 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 $\textbf{\textit{Quant}}^2 \longrightarrow \textbf{\textit{Quant}}$*

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

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

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

Note that we write $(Q\ T) \oplus \mathcal{R}$ to mean $(Q \oplus \mathcal{R})\ T$ and similarly $(Q\ T) \ominus \mathcal{R}$ to mean $(Q \ominus \mathcal{R})\ T$.

Definition 3. *We can consider a type environment* Γ *as a function* Identifiers $\to$ Types $\cup \{\bot\}$ *as follows:*

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

We write $\textbf{dom}(\Gamma)$ *to mean* $\{x \in \textsc{Identifiers} \mid \Gamma(x) \neq \bot\}$, *and* $\Gamma|_X$ *to mean the environment* $\{x : \tau \in \Gamma \mid x \in X\}$ *(restricting the domain of* Γ*).*

Definition 4. *Let* Q *and* $\mathcal{R}$ *be type quantities,* T_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*

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

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

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

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]**

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

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

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

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

[In implementation, we could implement records as a map from strings (the field name) to the values; the advantage being that we can make use of structural subtyping with no additional implementation issues. I imagine it's probably somewhat less efficient that doing a more standard offset-based approach though.]

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]**.

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

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

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

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

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).

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

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.

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

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

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

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

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.

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

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

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

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

$\boxed{\Gamma \vdash \mathcal{L} : \tau}$ **Non-consuming location typing** This judgment 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.]**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[TODO: Finish handling currying transformers.]

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

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

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

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

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

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

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

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

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

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

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

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

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.]**

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

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.]**

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

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.

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

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

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

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

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

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

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

$\boxed{\text{demote}(\tau) = \sigma}$ $\boxed{\text{demote}_*(T_1) = T_2}$ **Type Demotion** demote and demote$_*$ take a type and "strip" all the asset modifiers from it, as well as unfolding named type definitions. This process is useful, because it allows selecting asset types without actually having a value of the desired asset type. 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.

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

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

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

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

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

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

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

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

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

A.3 WIP ideas

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

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

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

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

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

[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:

```
{ 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.