Psamathe: A DSL with Flows for Safe Blockchain Assets

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

Michael Coblenz
University of Maryland
College Park, USA
mcoblenz@umd.edu

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, USA
jonathan.aldrich@cs.cmu.edu

ABSTRACT

Blockchains host smart contracts for crowdfunding, tokens, and many other purposes. Vulnerabilities in contracts are often discovered, leading to the loss of large quantities of money. Psamathe is a new language we are designing around a new flow abstraction, reducing asset bugs and making contracts more concise than in existing languages. We present an overview of Psamathe, including a partial formalization. We also discuss several example contracts in Psamathe, and compare the Psamathe examples to the same contracts written in Solidity.

1 INTRODUCTION

Blockchains are increasingly used as platforms for applications called *smart contracts* [?], which automatically manage transactions in an mutually agreed-upon way. Commonly proposed and implemented applications include supply chain management, healthcare, voting, crowdfunding, auctions, and more [? ? ?]. Smart contracts often manage *digital assets*, such as cryptocurrencies, or, depending on the application, bids in an auction, votes in an election, and so on. These contracts cannot be patched after deployment, even if security vulnerabilities are discovered. Some estimates suggest that as many as 46% of smart contracts may have vulnerabilities [?]. Vulnerabilities in smart contracts can lead to the loss of large quantities of money—the well-known DAO attack [?] caused the loss of over 40 million dollars.

Psamathe (/sɑmɑθi/) is a new programming language we are designing around *flows*, which are a new abstraction representing an atomic transfer operation. Together with features such as *modifiers*, flows provide a **concise** way to write contracts that **safely** manage assets (see Section 2). Solidity, the most commonly-used smart contract language on the Ethereum blockchain [?], does not provide analogous support for managing assets. Typical smart contracts are more concise in Psamathe than in Solidity, because Psamathe handles common patterns and pitfalls automatically. A formalization of Psamathe is in progress [?], with an *executable semantics* implemented in the 𝕂-framework [?], which is already capable of running the examples shown in Figures 1 and 5 (ERC-20 and a voting contract).

Other newly-proposed blockchain languages include Flint, Move, Nomos, Obsidian, and Scilla [? ? ? ? ?]. Scilla and Move are intermediate-level languages, whereas Psamathe is intended to be a high-level language. Obsidian, Move, Nomos, and Flint use linear or affine types to manage assets; Psamathe uses *type quantities*, which extend linear types to allow a more precise analysis of the flow of values in a program. None of the these languages have flows or provide support for all the modifiers that Psamathe does.

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

Figure 1: A Psamathe contract with a simple `transfer` function, 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 LANGUAGE

A Psamathe program is made of *transformers* and *type declarations*. Transformers contain *flows* describing the how values are transferred between variables. Type declarations provide a way to name types and to mark values with *modifiers*, such as **`asset`**.

Figure 1 shows a simple contract declaring a type and a transformer, which implements the core of ERC-20's `transfer` function. ERC-20 is a standard providing a bare-bones interface for token contracts managing *fungible* tokens. Fungible tokens are interchangeable (like most currencies), so it is only important how many tokens are owned by an entity, not **which** tokens.

2.1 Overview

Psamathe is built around the concept of a flow. Using the more declarative, *flow-based* approach provides the following advantages over imperative state updates:

- **Static safety guarantees**: Each flow is guaranteed to preserve the total amount of assets (except for flows that explicitly consume or allocate assets). The total amount of a nonconsumable asset never decreases. Each asset has exactly one reference to it, either via a variable in the current environment, or in a table/record. The **`immutable`** modifier prevents values from changing.
- **Dynamic safety guarantees**: Psamathe automatically inserts dynamic checks of a flow's validity; e.g., a flow of money would fail if there is not enough money in the source, or if there is too much in the destination (e.g., due to overflow). The **`unique`** modifier, which restrict values to never be created more than once, is also checked dynamically.
- **Data-flow tracking**: We hypothesize that flows provide a clearer way of specifying how resources flow in the code itself, which may be less apparent using other approaches, especially in complicated contracts. Additionally, developers must explicitly mark when assets are *consumed*, and only assets marked as **`consumable`** may be consumed.

```
1 var winners : list Ticket <-- tickets[nonempty st ticketWins(winNum, _)]
2 // Split jackpot among winners
3 winners --> payEach(jackpot / length(winners), _)
4 balance --> lotteryOwner.balance
5 // Lottery is over, destroy losing tickets
6 tickets --> consume
```

Figure 2: A code snippet that handles the process of ending a lottery.

- **Error messages**: When a flow fails, the Psamathe runtime provides automatic, descriptive error messages, such as

```
Cannot flow <amount> Token from account[<src>] to account[<dst>]:
    source only has <balance> Token.
```

  Flows enable such messages by encoding information into the source code.

Each variable and function parameter has a *type quantity*, approximating the number of values, which is one of: `empty`, `any`, `one`, `nonempty`, or `every`. Only `empty` asset variables may be dropped. Type quantities are inferred if omitted; every type quantity in Figure 1 can be omitted.

Modifiers can be used to place constraints on how values are managed: they are **`asset`**, **`consumable`**, **`fungible`**, **`unique`**, and **`immutable`**. An **`asset`** is a value that must not be reused or accidentally lost, such as money. A **`consumable`** value is an **`asset`** that it may be appropriate to dispose of, via the `consume` construct, documenting that the disposal is intentional. For example, while bids should not be lost **during** an auction, it is safe to dispose of them after the auction ends. A **`fungible`** value can be **merged**, and it is **not** **`unique`**. The modifiers **`unique`** and **`immutable`** provide the safety guarantees mentioned above.

We now give examples using modifiers and type quantities to guarantee additional correctness properties in the context of a lottery. The **`unique`** and **`immutable`** modifiers ensure users enter the lottery at most once, while **`asset`** ensures that we do not accidentally lose tickets. We use **`consumable`** because tickets no longer have any value when the lottery is over.

```
type TicketOwner is unique immutable address
type Ticket is consumable asset {
    owner : TicketOwner,
    guess : uint256
}
```

Consider the code snippet in Figure 2, handling ending the lottery. The lottery cannot end before there is a winning ticket, enforced by the `nonempty` in the *filter* on line 1; note that, as `winners` is `nonempty`, there cannot be a divide-by-zero error. Without line 4, Psamathe would give an error indicating `balance` has type `any ether`, not `empty ether`—a true error, because in the case that the jackpot cannot be evenly split between the winners, there will be some ether left over.

One could try automatically inserting dynamic checks in a language like Solidity, but in many cases it would require additional annotations. Such a system would essentially reimplement flows, providing some benefits of Psamathe, but not the same static guarantees. Some patchwork attempts already exist, such as the SafeMath library which checks for the specific case of underflow and overflow. For example, consider the following code snippet in Psamathe, which performs the task of selecting a user by some predicate `P`.

```
var user : User <-- users[one such that P(_)]
```

This line expresses that we wish to select exactly one user satisfying the predicate. There is no way to express this same constraint in Solidity (or most languages) without manually writing code to check it. Additionally, in Solidity, variables are initialized with default values, making uniqueness difficult to enforce.

Programs in Psamathe are *transactional*: 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, the error propagates, like an exception, until it either: a) reaches the top level, and the entire transaction fails; or b) reaches a **`catch`**, and then only the changes made in the corresponding **`try`** block will be reverted, and the code in the **`catch`** block will be executed.

Figure 3 shows the abstract syntax of the core calculus of Psamathe.

$$f \in \textsc{TransformerNames} \qquad t \in \textsc{TypeNames}$$
$$a, x, y, z \in \textsc{Identifiers} \qquad n, m \in \mathbb{N}$$

$\mathcal{Q}, \mathcal{R}, \mathcal{S}$	::=	$\texttt{one} \mid \texttt{any} \mid \texttt{nonempty} \mid \texttt{empty} \mid \texttt{every}$	(type quantities)
M	::=	$\texttt{fungible} \mid \texttt{unique} \mid \texttt{immutable}$	(modifiers)
	$\mid$	$\texttt{consumable} \mid \texttt{asset}$	(modifiers)
T	::=	$\texttt{bool} \mid \texttt{nat} \mid t \mid \texttt{table}(\overline{x})\ \{\overline{x : \tau}\}$	(base types)
τ, σ, π	::=	$\mathcal{Q}\ T$	(types)
$\mathcal{L}, \mathcal{K}$	::=	$\texttt{true} \mid \texttt{false} \mid n$	
	$\mid$	$x \mid \mathcal{L}.x \mid \texttt{var}\ x : T \mid [\overline{\mathcal{L}}] \mid \{\overline{x : \tau \mapsto \mathcal{L}}\}$	
	$\mid$	$\texttt{copy}(\mathcal{L}) \mid \texttt{zip}(\overline{\mathcal{L}})$	
	$\mid$	$\mathcal{L}[\mathcal{L}] \mid \mathcal{L}[\mathcal{Q}\ \text{s.t.}\ f(\overline{\mathcal{L}})] \mid \texttt{consume}$	
$\texttt{Trfm}$	::=	$\texttt{new}\ t(\overline{\mathcal{L}}) \mid f(\overline{\mathcal{L}})$	(transformer calls)
$\texttt{Stmt}$	::=	$\mathcal{L} \to \mathcal{L} \mid \mathcal{L} \to \texttt{Trfm} \to \mathcal{L}$	(flows)
	$\mid$	$\texttt{try}\ \{\overline{\texttt{Stmt}}\}\ \texttt{catch}\ \{\overline{\texttt{Stmt}}\}$	(try-catch)
$\texttt{Decl}$	::=	$\texttt{transformer}\ f(\overline{x : \tau}) \to x : \tau\ \{\overline{\texttt{Stmt}}\}$	(transformers)
	$\mid$	$\texttt{type}\ t\ \texttt{is}\ \overline{M}\ T$	(type decl.)
$\texttt{Prog}$	::=	$\overline{\texttt{Decl}}; \overline{\texttt{Stmt}}$	(programs)

Figure 3: Syntax of the core calculus of Psamathe.

2.2 Partial Formalization

We now present typing and evaluation rules for a fragment of the core calculus, describing the basics of flows. This fragment is the smallest fragment capable of defining complete, simple programs in Psamathe.

Statics. Below we show the type rules needed to check flows between variables. We use Γ and Δ as type environments, pairs of variables and types, identified with partial functions between the two.

Define $\# : \mathbb{N} \cup \{\infty\} \to Q$ so that $\#(n)$ is the best approximation by type quantity of n, i.e.,

$$\#(n) = \begin{cases} \texttt{empty} & \text{if } n = 0 \\ \texttt{one} & \text{if } n = 1 \\ \texttt{nonempty} & \text{if } n > 1 \\ \texttt{every} & \text{if } n = \infty \end{cases}$$

First, rules checking the types of the source and destination locators, and building up the appropriate updaters.

$\boxed{\Gamma \vdash_M f; \mathcal{L} : \tau \dashv \Delta}$ **Locator Typing** This judgement states in the environment Γ and *mode* M, using $\mathcal{L}$ according to the *updater* f will yield a value of type τ and the new type environment Δ.

A *mode* M is either S, meaning source, or D, meaning destination. This ensures that we don't use, for example, numeric literals as the destination of a flow. We refer to Γ as the *input environment* and Δ as the *output environment*. We use f to refer to a function on types ($\textsc{Type} \to \textsc{Type}$), called an *updater*. We call such functions *updaters*. We adopt the convention that if $f : \textsc{Type}^n \to \textsc{Type}$ and $g_1, \ldots, g_n : \textsc{Type} \to \textsc{Type}$, then $f(g_1, \ldots, g_n) : \textsc{Type} \to \textsc{Type}$, where

$$f(g_1, \ldots, g_n)(\tau) = f(g_1(\tau), \ldots, g_n(\tau))$$

We use the following type functions:

- $\mathbf{1}_{\textsc{Type}}$ is the identity function on types
- $\oplus Q, \ominus Q : \textsc{Type} \to \textsc{Type}$ are functions that take a type τ and add/subtract Q to/from its type quantity.
- $\text{with}_Q : \textsc{Type} \to \textsc{Type}$ is the function that replaces the type quantity of τ with Q; i.e., $\text{with}_Q(Q'\ T) = Q\ T$
- $\max : \textsc{Type}^2 \to \textsc{Type}$ returns the type with the larger type quantity
- $\sqcup : \textsc{Type}^2 \to \textsc{Type}$ performs the *join* of the two type quantities according to specificity (e.g., $\texttt{empty} \sqcup \texttt{nonempty} = \texttt{any}$)

[NOTE: I considered going with the below; I decided against it (for now) because some rules (e.g., FILTER) require "passing" the mode around $\boxed{\Gamma \vdash \overset{f}{\rightleftharpoons} \mathcal{L} : \tau \dashv \Delta}$]

[TODO: Write how this should be read (same for other judgements)]

Note that we write $\Gamma \vdash_M \overline{f; \mathcal{L} : \tau} \dashv \Delta$ where $|\overline{\mathcal{L}}| = n$ to mean for all $1 \le i \le n$, $\Gamma_i \vdash_M f_i; \mathcal{L}_i : \tau_i \dashv \Delta_i$ so that $\Gamma = \Gamma_1$, and $\Delta = \Delta_n$.

Constants of type **nat** or **bool** can only be used as sources—it doesn't make sense to flow values to a constant.

$$\frac{}{\Gamma \vdash_S f; n : \#(n)\ \texttt{nat} \dashv \Gamma}\ \textsc{Nat} \qquad \frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash_S f; b : \texttt{one bool} \dashv \Gamma}\ \textsc{Bool}$$

Variables may be used as either sources or destinations, as long as they are not immutable. We may also use them to select resources (in which case $f = \mathbf{1}_{\textsc{Type}}$), even if immutable.

$$\frac{f = \mathbf{1}_{\textsc{Type}} \text{ or } \neg(\tau\ \texttt{immutable})}{\Gamma, x : \tau \vdash_M f; x : \tau \dashv \Gamma, x : f(\tau)}\ \textsc{Var}$$

Variable definitions must be destinations, as newly defined variables are always empty, so there is no reason to use them as sources.

$$\frac{}{\Gamma \vdash_D f; (\texttt{var}\ x : T) : \texttt{empty}\ T \dashv \Gamma, x : f(\texttt{empty}\ T)}\ \textsc{VarDef}$$

If we want to leave the original located value unchanged, we may use $\texttt{copy}(\mathcal{L})$ to deep-copy whatever $\mathcal{L}$ locates. Because the original value will be unchanged, we use $\mathbf{1}_{\textsc{Type}}$ as the updater. Copied values have two restrictions: 1) they must only be sources, because there would be no way to refer to the values if used as a destination; and 2) they must be non-assets, because copying an asset is forbidden. For this reason, the resulting type of a copy is the *demoted* type of τ, which is never an asset.

$$\frac{\Gamma \vdash_S \mathbf{1}_{\textsc{Type}}; \mathcal{L} : \tau \dashv \Gamma}{\Gamma \vdash_S f; \texttt{copy}(\mathcal{L}) : \text{demote}(\tau) \dashv \Gamma}\ \textsc{Copy}$$

List literals are also only allowed to be used as sources; we use updateElem to modify the updater f as appropriate to work on the elements of the list, rather than the whole list.

$$\frac{\Gamma \vdash_S \text{updateElem}(f); \overline{\mathcal{L} : \tau} \dashv \Delta}{\Gamma \vdash_S f; [\tau; \overline{\mathcal{L}}] : \#(|\overline{\mathcal{L}}|)\ \texttt{table}(\cdot)\ \tau \dashv \Delta}\ \textsc{List}$$

where

$$\text{updateElem}(f) = \begin{cases} f & \text{if } f \in \{\mathbf{1}_{\textsc{Type}}, \text{with}_{\texttt{empty}}\} \\ (\mathbf{1}_{\textsc{Type}} \sqcup \text{with}_{\texttt{empty}}) & \text{otherwise} \end{cases}$$

Next, we consider record literals, which can also only be used as sources. We use updateElem again, because it captures the difference **[I think...]** between selecting the whole record and selecting parts of the fields of the record.

$$\frac{\Gamma \vdash_S \text{updateElem}(f); \overline{\mathcal{L} : \tau} \dashv \Delta}{\Gamma \vdash_S f; \left\{\overline{x : \tau \mapsto \mathcal{L}}\right\} : \texttt{one}\ \{\overline{x : \tau}\} \dashv \Delta}\ \textsc{Record}$$

The FILTER rule has several parts. We must ensure that the predicate, p, really is a predicate: it accepts values of type elemtype(T) and returns **one bool**. We next check that the arguments have the correct types. Next, we check that the location being filtered is of the right type, and updates its type, as appropriate. We use $\max(f, \ominus Q)$ to capture whether f is $\mathbf{1}_{\textsc{Type}}$ or performs some modification. For example, suppose $f(Q\ T) = \texttt{empty}\ T$. Then the intention is to select every located value—but we will only locate Q values, so we should only subtract Q values, and we will have $\max(f(\mathcal{R}\ T), (\mathcal{R}\ T) \ominus Q) = \max(\texttt{empty}\ T, (\mathcal{R}\ T) \ominus Q) = (\mathcal{R}\ T) \ominus Q$, as desired. Next, we add the condition that $\mathcal{R} \ge Q$, catching any flows that will obviously fail at runtime (e.g., $\mathcal{L}[\texttt{nonempty}\ \text{s.t.}\ p(\overline{\mathcal{K}})]$ where $\mathcal{L}$ is **empty**). This condition is not strictly necessary, as it would be caught by the dynamic check.

$$\frac{\begin{array}{c} \texttt{transformer}\ p(\overline{x : \tau}, y : \text{elemtype}(T)) \to \texttt{one bool}\ \{\overline{\texttt{Stmt}}\} \\ \Gamma \vdash_S \mathbf{1}_{\textsc{Type}}; \overline{\mathcal{K} : \tau} \dashv \Gamma \\ \Gamma \vdash_M \max(f, \ominus Q); \mathcal{L} : \mathcal{R}\ T \dashv \Delta \qquad \mathcal{R} \ge Q \end{array}}{\Gamma \vdash_M f; \mathcal{L}[Q\ \text{s.t.}\ p(\overline{\mathcal{K}})] : Q\ T \dashv \Delta}\ \textsc{Filter}$$

The SELECT rule allows us to use one locator to select parts of another. The locator $\mathcal{K}$ that is used to select part of $\mathcal{L}$ will not be modified; it is considered a source (i.e., it is checked using the mode S) for the purposes of this rule, because the values must already be present in the locations specified (e.g., it would make no sense to use a variable definition or **consume** as $\mathcal{K}$). Additionally, $\mathcal{K}$ may be the demoted version of $\mathcal{L}$; e.g., if $\mathcal{L} : Q$ **Token** where **Token** is the type of Token represented by **`nat`**, we can use a value of type **`nat`** to select tokens. We modify the updater f as in FILTER.

$$\dfrac{\begin{array}{c}\Gamma \vdash_S \mathbf{1}_{\textsc{Type}}; \mathcal{K} : Q\ T' \dashv \Gamma \\ \Gamma \vdash_M \max(f, \ominus Q); \mathcal{L} : \mathcal{R}\ T \dashv \Delta \\ \mathcal{R} \geq Q \qquad \text{demote}_*(T') = \text{demote}_*(T)\end{array}}{\Gamma \vdash_M f; \mathcal{L}[\mathcal{K}] : Q\ T \dashv \Delta}\ \textsc{Select}$$

We may only use **`consume`** as a destination, and only if T is a consumable type. We arbitrarily choose **`empty`** as the type quantity for **`consume`**.

$$\dfrac{T\ \texttt{consumable}}{\Gamma \vdash_D f; \texttt{consume} : \texttt{empty}\ T \dashv \Gamma}\ \textsc{Consume}$$

Finally, we may take any locator and type it with a less specific, but compatible, quantity. For example, if we know that $\mathcal{L}$: **`one`** T, then we can also say that $\mathcal{L}$: **`nonempty`** T or $\mathcal{L}$: **`any`** T; however, if we have $\mathcal{L}$: **`any`** T, we cannot say that $\mathcal{L}$: **`one`** T.

$$\dfrac{\Gamma \vdash_M f; \mathcal{L} : \mathcal{R}\ T \dashv \Delta \qquad \mathcal{R} \sqsubseteq Q}{\Gamma \vdash_M f; \mathcal{L} : Q\ T \dashv \Delta}\ \textsc{Subquant}$$

We now consider type checking statements.

$\boxed{\Gamma \vdash S\ \mathbf{ok} \dashv \Delta}$ **Statement Well-formedness** This judgement states that in the environment Γ, the statement S is well-formed and it transforms Γ into Δ.

We check that $\mathcal{L}$ and $\mathcal{K}$ have the same base type. All the values selected by $\mathcal{L}$ are cleared out and added to $\mathcal{K}$, using the updaters.

$$\dfrac{\begin{array}{c}\Gamma \vdash_S \text{with}_{\texttt{empty}}; \mathcal{L} : Q\ T \dashv \Delta \\ \Delta \vdash_D (\oplus Q); \mathcal{K} : \mathcal{R}\ T; \dashv \Xi\end{array}}{\Gamma \vdash (\mathcal{L} \to \mathcal{K})\ \mathbf{ok} \dashv \Xi}\ \textsc{Ok-Flow}$$

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

$$\dfrac{T\ \texttt{asset} \Rightarrow \texttt{asset} \in \overline{M}}{\vdash (\texttt{type}\ t\ \texttt{is}\ \overline{M}\ T)\ \mathbf{ok}}\ \textsc{Ok-Type}$$

$$\dfrac{\begin{array}{c}\overline{x : \tau}, y : \tau_y, z : \texttt{empty}\ T_z \vdash \overline{\texttt{Stmt}\ \mathbf{ok}} \dashv \Delta, \overline{x : \tau}, z : Q_z\ T_z \\ \forall v : \sigma \in \Delta. \neg(\sigma\ \texttt{asset})\end{array}}{(\texttt{transformer} f(\overline{x : \tau}, y : \tau_y) \to Q_z\ T_z\ \{\overline{\texttt{Stmt}}\})\ \mathbf{ok}}\ \textsc{Ok-Transformer}$$

Dynamics. Below are the rules to evaluate statements of flows between variables.

We introduce sorts for *values*, *resources*, values tagged with their type, and storage values. Storage values are either a natural number, indicating a location in the store, or amount(n), indicating n of some resource. Locators evaluate to storage value pairs, i.e., (ℓ, k), where ℓ indicates the parent location of the value, and k indicates which value to select from the parent location. If $\ell = k$, then every value should be selected. This is useful because it allows us to locate only part of a fungible resources, or a specific element inside a list. The **`select`**(ρ, ℓ, k) construct resolves storage value pairs into the resource that should be selected.

$$\begin{array}{llll} V & ::= & n \mid \texttt{error} & \text{(values)} \\ R & ::= & (T, V) & \text{(resources)} \\ \ell, k & ::= & n \mid \text{amount}(n) & \text{(storage values)} \\ \mathcal{L} & ::= & \ldots \mid (n, \ell) & \\ \texttt{Stmt} & ::= & \ldots \mid \texttt{revert} & \end{array}$$

DEFINITION 1. *An environment* Σ *is a tuple* (μ, ρ) *where* μ : *IDENTIFIERNAMES* $\rightharpoonup \mathbb{N} \times \ell$ *is the* variable lookup environment, *and* $\rho : \mathbb{N} \rightharpoonup R$ *is the* storage environment.

We now give rules for how to evaluate programs containing flows between natural numbers and variables.

$\boxed{\langle \Sigma, \mathcal{L} \rangle \to \langle \Sigma, \mathcal{L} \rangle}$ **Locator Evaluation** We begin with rules to evaluate locators. Note that $(\ell, \text{amount}(n))$ and (ℓ, ℓ) are equivalent w.r.t. **`select`** when $\rho(\ell) = (T, n)$ for some fungible T.

$$\dfrac{m \notin \textbf{dom}(\rho)}{\langle \Sigma, n \rangle \to \langle \Sigma[\rho \mapsto \rho[\ell \mapsto (\texttt{nat}, n)]], (m, \text{amount}(n)) \rangle}\ \textsc{Loc-Nat}$$

$$\dfrac{n \notin \textbf{dom}(\rho)}{\langle \Sigma, b \rangle \to \langle \Sigma[\rho \mapsto \rho[\ell \mapsto (\texttt{bool}, b)]], (n, n) \rangle}\ \textsc{Loc-Bool}$$

$$\dfrac{\mu(x) \neq \bot}{\langle \Sigma, x \rangle \to \langle \Sigma, \mu(x) \rangle}\ \textsc{Loc-Id}$$

$$\dfrac{n \notin \textbf{dom}(\rho)}{\langle \Sigma, \texttt{var}\ x : T \rangle \to \langle (\mu[x \mapsto \ell], \rho[\ell \mapsto \text{empty}(T)]), (n, n) \rangle}\ \textsc{Loc-VarDef}$$

$$\dfrac{\langle \Sigma, \mathcal{L} \rangle \to \left\langle \Sigma', \mathcal{L}' \right\rangle}{\left\langle \Sigma, [\tau; \overline{(n, k)}, \mathcal{L}, \overline{\mathcal{K}}] \right\rangle \to \left\langle \Sigma', [\tau; \overline{(n, k)}, \mathcal{L}', \overline{\mathcal{K}}] \right\rangle}\ \textsc{Loc-List}$$

$\boxed{\langle \Sigma, \overline{\texttt{Stmt}} \rangle \to \langle \Sigma, \overline{\texttt{Stmt}} \rangle}$ **Statement Evaluation** Finally, the rule for the flows. We first resolve the selected resources, then subtract them from their parent locations, and finally add them all to the destination location.

$$\dfrac{\texttt{select}(\rho, n, \ell) = R \qquad \rho(m) + R \neq \texttt{error}}{\langle \Sigma, (n, \ell) \to (m, k) \rangle \to \Sigma[\rho \mapsto \rho[n \mapsto \rho(n) - R, m \mapsto \rho(m) + R]]}\ \textsc{Flow}$$

$$\dfrac{\texttt{select}(\rho, n, \ell) = R \qquad \rho(m) + R = \texttt{error}}{\langle \Sigma, (n, \ell) \to (m, k) \rangle \to \langle \Sigma, \texttt{revert} \rangle}\ \textsc{Flow-Error}$$

$$\dfrac{\overline{\texttt{select}(\rho, n, \ell) = R}}{\left\langle \Sigma, [\tau; \overline{(n, \ell)}] \to (m, k) \right\rangle \to \Sigma[\rho \mapsto \rho[n \mapsto \rho(n) - R, m \mapsto \rho(m) + R]]}\ \textsc{Flow-List}$$

[TODO: Need to finish FLOW-LIST (needs to prperly select and remove and add)]

$\boxed{\Gamma \leftrightarrow \Sigma}$ **Environment compatibility** **[This is only necessary for the proofs]** We check that every variable in Γ appears in the variable lookup environment μ and vice versa. Additionally, every variable x in the variable lookup environment must map to some value in the storage environment, which must be compatible with the type of x in Γ; that is, the *exact type* of **`select`**$(\rho, \mu(x))$ must be at least as specific as the type of x in the environment.

```
mapping (address => uint256) balances;
function transfer(address dst, uint256 amount) public {
  require(amount <= balances[msg.sender]);
  balances[msg.sender] = balances[msg.sender].sub(amount);
  balances[dst] = balances[dst].add(amount);
}
```

Figure 4: An implementation of ERC-20's `transfer` function in Solidity from one of the reference implementations [?]. All preconditions are checked manually. Note that we must include the `SafeMath` library (not shown) to use the `add` and `sub` functions, which check for underflow/overflow.

$$\frac{\textbf{dom}(\Gamma) = \textbf{dom}(\mu) \qquad \forall x : \tau \in \Gamma.\text{exactType}(\textbf{select}(\rho, \mu(x))) \sqsubseteq_\tau \tau}{\Gamma \leftrightarrow \Sigma}\ \textsc{Compat}$$

where $Q\ T \sqsubseteq_\tau \mathcal{R}\ T' \Leftrightarrow Q \sqsubseteq \mathcal{R}$ and $T = T'$.

3 EXAMPLES

In this section, we present additional examples, showing that Psamathe and flows are useful for a variety of smart contracts. We also show examples of these same contracts in Solidity, and compare the Psamathe implementations to those in Solidity.

3.1 ERC-20 in Solidity

Each ERC-20 contract manages the "bank accounts" for its own tokens, keeping track of how many tokens each account has; accounts are identified by addresses. We compare the Psamathe implementation in Figure 1 to Figure 4, which shows a Solidity implementation of the same function. In this case, the sender's balance must be at least as large as `amount`, and the destination's balance must not overflow when it receives the tokens. Psamathe automatically inserts code checking these two conditions, ensuring the checks are not forgotten. As noted above, we can automatically generate descriptive error messages with no additional code, which are not present in the Solidity implementation.

3.2 Voting

One proposed use for blockchains is for voting [?]. Figure 5 shows the core of an implementation of a voting contract in Psamathe. Each contract instance has several proposals, and users must be given permission to vote by the chairperson, assigned in the constructor of the contract (not shown). Each eligible voter can vote exactly once for exactly one proposal, and the proposal with the most votes wins. This example shows some uses of the **`unique`** modifier; in this contract, **`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. This example show that Psamathe, as well as flows, are suited to a wide range of common smart contract applications.

Figure 6 shows an implementation of the same voting contract in Solidity, based on the Solidity by Example tutorial [?]. Again, we must manually check all preconditions.

```
type Voter is unique immutable asset address
type ProposalName is unique immutable asset string
type Election is asset {
  chairperson : address,
  eligibleVoters : set Voter,
  proposals : map ProposalName => set Voter
}
transformer giveRightToVote(this : Election, voter : address) {
  only when msg.sender = this.chairperson
  new Voter(voter) --> this.eligibleVoters
}
transformer vote(this : Election, proposal : string) {
  this.eligibleVoters --[ msg.sender ]-> this.proposals[proposal]
}
```

Figure 5: A simple voting contract in Psamathe.

```
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, "No right to vote");
    require(!sender.voted, "Already voted.");
    sender.voted = true;
    sender.vote = proposal;
    proposals[proposal].voteCount += sender.weight;
  }
}
```

Figure 6: A simple voting contract in Solidity.

3.3 Blind Auction

Another proposed use of blockchains is auctions [?]. Figure 7 shows an implementation of the *reveal phase* of a *blind auction* in Psamathe. A blind auction is an auction in which bids are placed, but not revealed until the auction has ended, meaning that other bidders have no way of knowing what bids have been placed so far. Because transactions on the Ethereum blockchain are publicly viewable, the bids must be blinded cryptographically, in this case, using the KECCAK-256 algorithm [?]. Bidders sent the hashed bytes of their bid, that is, the value (in ether) and some secret string of bytes, along with a deposit of ether, which must be at least as large as the intended value of the bid for the bid to be valid. After bidding is over, they must *reveal* their bid by sending a transaction

```
1  type Bid is consumable asset {
2    sender : address,
3    blindedBid : bytes,
4    deposit : ether
5  }
6  type Reveal is { value : nat, secret : bytes }
7  type Auction is asset {
8    biddingEnd : nat, revealEnd : nat, ended : bool,
9    bids : map address => list Bid,
10   highestBidder : address, highestBid : ether,
11   pendingReturns : map address => ether
12 }
13 transformer reveal(this : Auction, reveals : list Reveal) {
14   only when biddingEnd <= now and now <= revealEnd
15   zip(this.bids[msg.sender], reveals)
16     --[ any such that _.fst.blindedBid = keccak256(_.snd) ]
17     --> this.revealBid(_.fst, _.snd)
18 }
19 transformer revealBid(this : Auction, bid : Bid, reveal : Reveal) {
20   try {
21     only when reveal.value >= this.highestBid
22     this.highestBid --> this.pendingReturns[highestBidder]
23     bid.deposit --[ reveal.value ]-> this.highestBid
24     bid.sender --> this.highestBidder
25   } catch {}
26   bid.deposit --> bid.sender.balance
27   bid --> consume
28 }
```

Figure 7: Implementation of reveal phase of a blind auction contract in Psamathe.

containing these details, which will be checked by the `Auction` contract (line 16). Any extra value in the bid (used to mask the true value of the bid), will be returned to the bidder.

This example uses a pipeline of locators and transformers (lines 15-17) to concisely process each revealed bid, showing another case in which flows provide a clean way to write smart contracts.

4 CONCLUSION AND FUTURE WORK

We have presented the Psamathe language for writing safer smart contracts. Psamathe uses the new flow abstraction, assets, and type quantities to provide its safety guarantees. We have shown example smart contracts in both Psamathe and Solidity, 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 prove its safety properties. We also hope to study the benefits and costs of the language via case studies, performance evaluation, and the application of flows to other domains. Finally, we would also like to conduct a user study to evaluate the usability of the flow abstraction and the design of the language, and to compare it to Solidity, which we hypothesize will show that developers write contracts with fewer asset management errors in Psamathe than in Solidity.

[TODO: For some reason the references stopped showing up...???]