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 [17], 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 [9–11]. 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 [12]. Vulnerabilities in smart contracts can lead to the loss of large quantities of money—the well-known DAO attack [16] caused the loss of over 40 million dollars.

Psamathe (/psa-ma-the/) 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 [1], 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 [2], with an executable semantics implemented in the X-framework [13], 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 [6–8, 14, 15]. 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.

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 non-consumable 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.
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 may be appropriately disposed 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.

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.

1. **Partial Formalization**

   **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.

   The functions u are u are updaters, specifically, \( u, v \in (\Gamma \cup (\Gamma \times (\tau \rightarrow \tau)) \rightarrow \Gamma) \), functions that modify an environment given some
function on types. Define
\[
\text{nat} = \begin{cases} 
\bot & \text{if } T \text{ immutable} \\
\mu & \text{otherwise}
\end{cases}
\]

Define \# : \mathbb{N} \cup \{\infty\} \to Q so that \#(n) is the best approximation by type quantity of \( n \), i.e.,
\[
\#(n) = \begin{cases} 
\text{empty} & \text{if } n = 0 \\
\text{one} & \text{if } n = 1 \\
\text{nonempty} & \text{if } n > 1 \\
\text{every} & \text{if } n = \infty
\end{cases}
\]

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

\[\Gamma \vdash M (L : \tau): u\]

**Locator Typing** Here \( M \) is a mode, either \( S \), meaning source, \( D \), meaning destination, or \( * \), meaning either mode is acceptable. This ensures that we don’t use, for example, numeric literals as the destination of a flow.

\[
\frac{\Gamma : S \vdash n : \#(n) \text{ nat}; (\Delta, f) \rightarrow \Delta}{\Gamma, x : Q T \vdash (x : Q T); ((\Delta, f) \rightarrow \Delta[x \mapsto f(\Delta(x))] )|| T} \quad \text{VAR}
\]

\[
\frac{\Gamma : D \vdash (\langle \text{var } x : T : \text{empty} \rangle; (\Delta, f) \rightarrow \Delta[x \mapsto f(\text{empty } T)] )}{\Gamma, \text{var } x : T \vdash (x : Q T); ((\Delta, f) \rightarrow \Delta[x \mapsto f(\text{empty } T)] )|| T} \quad \text{VARDef}
\]

Below is the type rule checking whether a flow (without a transformer) is valid.

\[\Gamma \vdash S \text{ ok } \Delta\]

**Statement Well-formedness** Here \( \Gamma \) is the input environment and \( \Delta \) is the output environment, used to track the flow of resources throughout a sequence of statements.

In Ok-Flow, Checking \( u \neq \bot \) and \( v \neq \bot \) ensures that \( L \) and \( K \) are valid sources/destinations of the flow (e.g., not immutable). We check that \( L \) and \( K \) have the same base type. We then clear out all the values selected by \( L \), using \( u \), and add them to \( K \), using \( v \).

\[
\frac{\Gamma : S \vdash L : Q T; u \quad \Delta = u(\Gamma, (Q' T')) \rightarrow \text{empty } T'}{\Delta \vdash D : K : R T; v \quad u \neq \bot \quad v \neq \bot}{\Gamma \vdash (L \rightarrow K) \text{ ok } \Delta \vdash \langle \text{var } \Delta, \tau \mapsto \tau \circ Q \rangle} \quad \text{OK-Flow}
\]

**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., \((\ell, k)\), where \( \ell \) 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 \textbf{select}(\rho, \ell, k)\) construct resolves storage value pairs into the resource that should be selected.

\[
\begin{align*}
V & \equiv n \quad (\text{values}) \\
R & \equiv (T, V) \quad (\text{resources}) \\
\ell, k & \equiv n | \text{amount}(n) \quad (\text{storage values}) \\
L & \equiv \ldots | (\ell, k)
\end{align*}
\]

**Definition 1.** An environment \( \Sigma \) is a tuple \((\mu, \rho)\) where \( \mu : \text{IDENTIFIER}\text{NAMES} \to \mathbb{N} \times \ell \) is the variable lookup environment, and \( \rho : \mathbb{N} \to R \) is the storage environment.

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

\[\text{Loc-Nat} \quad \begin{array}{c}
\Sigma, n \rightarrow \langle \Sigma[\rho \mapsto \rho[\ell \mapsto (\text{nat}, n)]], (\ell, \text{amount}(n))\rangle
\end{array}\]

\[\text{Loc-Id} \quad \begin{array}{c}
\Sigma, x \rightarrow \langle \Sigma, (\mu(x), \mu(x))\rangle
\end{array}\]

\[\text{Loc-VarDef} \quad \begin{array}{c}
\Sigma, \text{var } x : T \rightarrow \langle \Sigma[\mu \mapsto \mu[x \mapsto \ell], \rho \mapsto \rho[\ell \mapsto \text{empty } T]\rangle, (\ell, t)\rangle
\end{array}\]

**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.

\[\text{Flow} \quad \begin{array}{c}
\text{select}(\rho, \ell, k) = R \\
\langle \Sigma, \text{select}(\rho, \ell, k) \rightarrow (i, j) \rangle \rightarrow \Sigma[\rho \mapsto \rho[\ell \mapsto \rho(\ell - R, j \mapsto \rho(j) + \sum R)]\]
\end{array}\]

### 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 [9]. 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
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 [4]. 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.

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.

Figure 6: A simple voting contract in Solidity.

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.

3.3 Blind Auction

Another proposed use of blockchains is auctions [9]. 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 [5]. 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 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.

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.

REFERENCES

[1] Ethereum for developers (2020), https://ethereum.org/en/developers/
[2] Psamathe (Oct 2020), https://github.com/ReedOei/Psamathe
[3] Solidity by example (2020), https://solidity.readthedocs.io/en/v0.7.0/solidity-by-example.html
[4] Tokens (Aug 2020), https://github.com/ConsenSys/Tokens
[5] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 313–314. Springer (2013)
[6] Blakeshe, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, D.R., Sezer, S., et al.: Move: A language with programmable resources (2019)
type Bid is consumable asset {
  sender : address,
  blindedBid : bytes,
  deposit : ether
}

type Reveal is { value : nat, secret : bytes }

type Auction is asset {
  biddingEnd : nat, revealEnd : nat, ended : bool,
  bids : map address => list Bid,
  highestBidder : address, highestBid : ether,
  pendingReturns : map address => ether
}

transformer reveal(this : Auction, reveals : list Reveal) {
  only when biddingEnd <= now and now <= revealEnd
  zip(this.bids[msg.sender], reveals)
    --[ any such that _.fst.blindedBid = keccak256(_.snd) ]
    --> this.revealBid(_.fst, _.snd)
}

transformer revealBid(this : Auction, bid : Bid, reveal : Reveal) {
  try {
    only when reveal.value >= this.highestBid
    this.highestBid --> this.pendingReturns[highestBidder]
    bid.deposit -->[ reveal.value ]-> this.highestBid
    bid.sender --> this.highestBidder
  } catch {}
  bid.deposit --> bid.sender.balance
  bid --> consume
}

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