Psamathe: A DSL for Safe Blockchain Assets

Reed Oei[1]

University of Illinois
reedoei2@illinois.edu

Abstract.

1 Introduction

Blockchains are increasingly used as platforms for applications called *smart contracts* [17], which automatically manage transactions in an unbiased, mutually agreed-upon way. Commonly proposed and implemented smart contracts include supply chain management [11], healthcare [10], *token contracts*, voting, crowdfunding, auctions, and more [9]. A *token contract* is a contract implementing a *token standard*, such as ERC-20 [1]. 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. Token contracts are common on the Ethereum blockchain [18]—about 73% of high-activity contracts are token contracts [13]. Smart contracts cannot be patched after being deployed, even if a security vulnerability is discovered. Some estimates suggest that as much as 46% of smart contracts may have some vulnerability [12].

Psamathe (/sɑmɑθi/) is a new programming language we are designing around a new abstraction, a *flow*, representing an atomic transfer operation. Together with *modifiers* and *locators*, flows provide a **concise** way to write smart contracts that **safely** manage assets.

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

Fig. 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.

Solidity is the most commonly-used smart contract language on the Ethereum blockchain [2], and 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.

Other many newly-proposed blockchain languages include Flint, Move, Nomos, Obsidian, and Scilla [15,6,8,7,16]. Scilla and Move are intermediate-level languages, whereas Psamathe is a high-level language. Obsidian, Move, Nomos, and Flint use linear or affine types to manage assets; Psamathe uses *type quantities*, which provide the benefits of *linear types*, but give a more precise analysis of the flow of values in a program. None of the these languages have flows, provide support for all the modifiers that Psamathe does, or have locators. In particular, the latter means that programs in those languages using linear or affine types must be more verbose to maintain linearity.

2 Language

A Psamathe program is made of *transformers* and *type declarations*. Transformers contain statements describing the flow of values between the variables of the transformer. Type declarations provide a way to mark values with *modifiers*. 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 for token contracts managing **fungible** tokens, and provides a bare-bones interface for this purpose.

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

– **Asset retention**: Because of the design of the language, each flow is guaranteed to preserve the total amount of assets (except for flows that consume or allocate assets), removing the need to verify such properties.
– **Precondition checking**: 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).
– **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.
– **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.
```

Flows enable such messages by encoding all the necessary information into the program.

Each variable and function parameter has a *type quantity*, approximating the number of values in the variable, which is one of: `empty`, `any`, !, `nonempty`, `every` ("!" means "exactly one"). Type quantities are inferred if omitted. Only `empty` asset variables may be dropped.

Modifiers can be used to place constraints on how values are managed: `asset`, `fungible`, `unique`, `immutable`, and `consumable`. An `asset` is a value that must not be reused or accidentally lost. A `fungible` value represents a quantity that can be **merged**, and it is **not** `unique`. For example, ERC-20 tokens are `fungible`. A `immutable` value cannot be changed; in particular, it cannot be the source or destination of a flow. A `unique` value only exists in at most one variable; it must be `immutable` and an `asset` to ensure it is not duplicated. ERC-721 tokens are `unique` and `immutable`. A `consumable` value is an `asset` that it may be appropriate to dispose of, done via the *`consume`* construct, documenting that the disposal is intentional.

[TODO: Discuss locators] [*Locators* are expressions that allow accessing or modifying parts of complex structures while maintaining linearity.]

Psamathe 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, 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.

A formalization of Psamathe is in progress [3], with an *executable semantics* implemented in the 𝕂-framework [14] capable of running the ERC-20 **[and hopefully with minimal more work, Voting]** examples discussed below.

3 Examples

3.1 ERC-20

Each ERC-20 contract manages the “bank accounts” for its own tokens, keeping track of how many tokens each user has; users are represented by addresses. Figure 2 shows a Solidity implementation of the ERC-20 function transfer (cf.

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

Fig. 2: An implementation of ERC-20’s transfer function in Solidity from one of the reference implementations [5]. 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.

Figure 1). This example shows the advantages of flows in precondition checking, data-flow tracking, and error messages. 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 that the checks are not forgotten.

3.2 Voting

```
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;
  }
}
```

Fig. 3: A simple voting contract in Solidity.

One proposed use for blockchains is for voting [9]. Figures 3 and 4 show the core of an implementation of a voting contract in Solidity and Psamathe, respectively, based on the Solidity by Example tutorial [4]. 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 user can vote exactly once for exactly one proposal, and the proposal with the most votes wins.

This example shows Psamathe is suited for a range of applications. It also 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.

```
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 = chairperson
    new Voter(voter) --> this.eligibleVoters
}
transformer vote(this : Election, proposal : string) {
    this.eligibleVoters --[ msg.sender ]-> this.proposals[proposal]
}
```

Fig. 4: A simple voting contract in Psamathe.

3.3 Blind Auction

[TODO: Describe, simplify]

4 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 shown two 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 implement the Psamathe language, and prove its safety properties. We also hope to study the benefits 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. Eip 20: Erc-20 token standard (2015), https://eips.ethereum.org/EIPS/eip-20
2. Ethereum for developers (2020), https://ethereum.org/en/developers/
3. Psamathe (Aug 2020), https://github.com/ReedOei/Psamathe
4. Solidity by example (2020), https://solidity.readthedocs.io/en/v0.7.0/solidity-by-example.html
5. Tokens (Aug 2020), https://github.com/ConsenSys/Tokens

```
type Bid is consumable asset {
  sender : address,
  blindedBid : bytes,
  deposit : ether
}
type Reveal is { value : nat, fake : bool, secret : bytes }
type BlindAuction is asset {
  beneficiary : address,
  biddingEnd : nat,
  revealEnd : nat,
  ended : bool,
  bids : map address => list Bid,
  highestBidder : address,
  highestBid : ether,
  pendingReturns : map address => ether
}

transformer bid(this : BlindAuction, bid : bytes) {
  only when now <= this.biddingEnd
  [ new Bid(msg.sender, big, msg.value) ] --> this.bids[msg.sender]
}
transformer reveal(this : BlindAuction, reveals : list Reveal) {
  only when biddingEnd <= now and now <= revealEnd
  zip(this.bids[msg.sender], reveals)
    --[ any such that _.fst.blindedBid = keccak256(_.snd) ]
    --> revealBid(this, _)
    --> placeBid(this, _)
}
transformer revealBid(this : BlindAuction, bid : Bid, reveal : Reveal)
  -> toPlace : list { sender : address, value : ether } {
  try {
    only when not reveal.fake
          and reveal.value >= this.highestBid
    bid.deposit --[ reveal.value ]-> var value : ether
    [ { sender -> bid.sender, value -> value } ] --> toPlace
  } catch {}
  bid.deposit --> bid.sender.balance
  bid --> consume
}
transformer placeBid(this : BlindAuction, toPlace : { sender : address, value : ether }) {
  try {
    only when highestBidder != 0x0
    this.highestBid --> this.pendingReturns[highestBidder]
  } catch { }
  toPlace.sender --> highestBidder
  toPlace.value --> highestBid
}
transformer withdraw(this : BlindAuction) {
  this.pendingReturns[msg.sender] --> msg.sender.balance
}
transformer auctionEnd(this : BlindAuction) {
  only when now >= this.revealEnd and not this.ended
  true --> this.ended
  highestBid --> beneficiary.balance
}
```

Fig. 5: **[TODO]**

6. Blackshear, 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)
7. Coblenz, M., Oei, R., Etzel, T., Koronkevich, P., Baker, M., Bloem, Y., Myers, B.A., Sunshine, J., Aldrich, J.: Obsidian: Typestate and assets for safer blockchain programming (2019)
8. Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. arXiv preprint arXiv:1902.06056 (2019)
9. Elsden, C., Manohar, A., Briggs, J., Harding, M., Speed, C., Vines, J.: Making sense of blockchain applications: A typology for hci. In: CHI Conference on Human Factors in Computing Systems. pp. 1–14. CHI '18 (2018). https://doi.org/10.1145/3173574.3174032, `http://doi.acm.org/10.1145/3173574.3174032`
10. Harvard Business Review: The potential for blockchain to transform electronic health records (2017), `https://hbr.org/2017/03/the-potential-for-blockchain-to-transform-electronic-health-records`
11. IBM: Blockchain for supply chain (2019), `https://www.ibm.com/blockchain/supply-chain/`
12. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. p. 254–269. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978309, `https://doi.org/10.1145/2976749.2978309`
13. Oliva, G., Hassan, A.E., Jiang, Z.: 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
14. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
15. Schrans, F., Eisenbach, S., Drossopoulou, S.: Writing safe smart contracts in flint. In: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming. pp. 218–219 (2018)
16. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with scilla. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360611, `https://doi.org/10.1145/3360611`
17. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997). https://doi.org/https://doi.org/10.5210/fm.v2i9.548
18. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)