Psamathe: A DSL for Safe Blockchain Assets

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

Abstract

Blockchains host smart contracts for voting, tokens, and other purposes. Vulnerabilities in contracts are common, often leading to the loss 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, and discuss two example contracts in Psamathe and Solidity.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**.

Keywords: domain specific languages, smart contracts

ACM Reference Format:
Reed Oei. 2020. Psamathe: A DSL for Safe Blockchain Assets. In *Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '20), November 15–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3426430.3428131

1 Introduction

Blockchains are increasingly used as platforms for applications called *smart contracts* [14], which automatically manage transactions. Commonly proposed and implemented contracts include supply chain management [10], healthcare [9], voting, crowdfunding, auctions, and more [8]. However, smart contracts cannot be patched after being deployed, even if a security vulnerability is discovered. The well-known DAO attack caused the loss of over 40 million dollars [13].

Psamathe (/sɑmɑθi/) is a new programming language we are designing to write safer contracts, focused on a new abstraction, a *flow*, representing an atomic transfer. The Psamathe language will also provide features to mark types with *modifiers*, such as **asset**, which combine with flows to make some classes of bugs impossible.

Solidity is the most commonly-used contract language on Ethereum [2], and does not provide analogous support for managing assets. Typical contracts are more **concise** in Psamathe than in Solidity, because Psamathe handles common patterns and pitfalls automatically.

Other newly-proposed blockchain languages include Flint, Move, Nomos, Obsidian, and Scilla [5–7, 11, 12]. Scilla and Move are intermediate 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 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 *contracts*, each containing *fields*, *types*, and *functions*. Each contract instance in Psamathe represents a contract on the blockchain, and the fields provide persistent storage. Figure 1 shows a simple contract implementing the core of ERC-20's `transfer` function. ERC-20 [1] is a standard for managing *fungible* tokens, which are interchangeable, like most currencies.

Psamathe is built around *flows*. Using the more declarative, *flow-based* approach provides the following advantages: Flows have a *source*, a *destination*, and an optional *selector*. When executed, every value selected (everything, by default) in the source is removed and combined with the destination.

- **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 in the source.
- **Data-flow tracking**: We hypothesize that flows provide a clearer way of specifying how resources flow, which may be less apparent using other approaches, especially in complicated contracts.

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

Figure 1. A Psamathe contract with a `transfer` function, transferring `amount` tokens from the sender's account to the destination account. It uses a single flow, which checks all the preconditions to ensure the transfer is valid.

SPLASH Companion '20, November 15–20, 2020, Virtual, USA

ACM ISBN 978-1-4503-8179-6/20/11...$15.00
https://doi.org/10.1145/3426430.3428131

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

Figure 2. A Solidity implementation of ERC-20's transfer, from a reference implementation [4]. Preconditions are checked manually. We must include the SafeMath library (not shown) to use add and sub (to check for under/over-flow).

- **Error messages**: When a flow fails, Psamathe provides automatic, descriptive error messages, using information in the code of the flow, such as:
  Cannot flow <amount> Token from account[<src>] to account[<dst>]: source only has <balance> Token.

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

Modifiers can be used to place constraints on how values are managed: **asset**, **fungible**, and **unique**. An **asset** is a value that must not be reused or accidentally lost. A **fungible** value represents an interchangeable value that can be **merged**. ERC-20 tokens are **fungible**. A **unique** value only exists in at most one variable, enforced by a dynamic check when created; it must be an **asset** to prevent duplication.

3 Examples

3.1 ERC-20

Figure 2 shows a Solidity implementation of the ERC-20 function transfer (cf. 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.

```
1  contract Ballot {
2      type Voter is unique asset address
3      type ProposalName is unique asset string
4      chairperson : address
5      voters : set Voter
6      proposals : map ProposalName => set Voter
7      transaction giveRightToVote(voter: address):
8          only when msg.sender = chairperson
9          new Voter(voter) --> voters
10     transaction vote(proposal: string):
11         voters --[ msg.sender ]-> proposals[proposal]
12 }
```

Figure 3. A simple voting contract in Psamathe.

3.2 Voting

Figures 3 and 4 show the core of a voting contract in Psamathe and Solidity, respectively, based on the Solidity by Example tutorial [3]. Each contract instance has several proposals, and a chairperson, assigned in the constructor (not shown), gives users permission to vote. Each user can vote exactly once for exactly one proposal. Note that values of type address are used to *select* Voters on line 11. We can do this because the *underlying type* of Voter is address—allowing us to refer to assets without the asset itself.

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 **asset** ensures that votes are not lost or double-counted.

4 Conclusion and Future Work

Psamathe is a language for safer contracts. Psamathe uses the new flow abstraction and modifiers (e.g., **asset**) to provide safety guarantees. We showed examples of contracts in Solidity and Psamathe, showing that Psamathe can express common contracts concisely, retaining key safety properties.

In the future, we plan to implement the Psamathe language, and prove its safety properties. We 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.

```
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 4. A simple voting contract in Solidity.

References

[1] 2015. EIP 20: ERC-20 Token Standard. Retrieved 2020-07-28 from https://eips.ethereum.org/EIPS/eip-20

[2] 2020. Ethereum for Developers. Retrieved 2020-07-31 from https://ethereum.org/en/developers/

[3] 2020. Solidity by Example. Retrieved 2020-07-28 from https://solidity.readthedocs.io/en/v0.7.0/solidity-by-example.html

[4] 2020. Tokens. Retrieved 2020-08-03 from https://github.com/ConsenSys/Tokens

[5] 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.

[6] 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

[7] 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).

[8] 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

[9] 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

[10] IBM. 2019. Blockchain for supply chain. Retrieved March 31, 2019 from https://www.ibm.com/blockchain/supply-chain/

[11] 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.

[12] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 185 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360611

[13] 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/

[14] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (1997). https://doi.org/10.5210/fm.v2i9.548