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, 3 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 for the blockchain often revolve around managing *digital assets*, such as smart contracts for supply chain management [10], healthcare [9], and other applications which require careful management of their respective assets such as voting, crowdfunding, or auctions [8]. One of the most common is a *token contract*—about 73% of high-activity contracts are token contracts [12].

Smart contracts cannot be patched after being deployed, even if a security vulnerability is discovered. Therefore, developers must carefully review contracts before deployment, 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 [15] 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 [11].

Psamathe (/sɑmɑθi/) is a new programming language we are designing around a new abstraction, called a *flow*, representing an atomic transfer operation, which is widely applicable to 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 certain values as *assets*, with various *modifiers* to control their use, which combine with flows to make certain classes of bugs impossible. Solidity, the most commonly-used language for writing smart contracts on the Ethereum blockchain [4], 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 *fields*, *types*, and *functions*. Each contract instance in Psamathe represents a contract on the blockchain, whose storage is represented by the fields. Type declarations allow annotating values with modifiers, such as **asset**.

Figure 1 shows a simple contract declaring 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.

Psamathe is built around the concept of a *flow*, an atomic, state-changing operation describing the transfer of values.

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**: Psamathe automatically checks that a flow is valid; e.g., a flow of money would fail if there is not enough in the source, or if there is too much in the destination (e.g., due to overflow).
- **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.
- **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 these messages by encoding the necessary information, instead of using low-level operations like increment and decrement.

Each variable in Psamathe has a *type quantity* at every point in the program, providing an approximation of the number of values in the variable. The list of allowed type quantities is: **empty**, **any**, !, **nonempty**, **every** ("!" means "exactly one"). Only **empty** asset variables may be dropped. Type quantities are **not** required in the surface language, and they will be added automatically if omitted. The type quantity system provides the benefits of *linear types*, but gives a more precise analysis of the flow of values in a program.

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

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

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. For example, ERC-20 tokens are `fungible`, while ERC-721 tokens are `unique` and `immutable`.

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, 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.

3 EXAMPLES

The full Solidity and Psamathe implementations can be found in our repository [1].

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. 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.

```
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 3: 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.

Figure 3 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 Voting

One proposed use for the blockchain is smart contracts for managing voting [8]. Figures 2a and 2b 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 2. 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.

4 RELATED WORK

There have been many new blockchain languages proposed, such as Flint [13], Move [5], Nomos [7], Obsidian [6], and Scilla [14]. 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, and none 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.

Solidity is the most common blockchain language, but does not have any features for managing assets.

5 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 studies on the benefits of the language such as case studies, evaluating its efficiency, and exploring the applicability of flows to other domains. Finally, we would also like to perform a user study to evaluate the usability of the flow abstraction and the design of the language, 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] 2020. Ethereum for Developers. Retrieved 2020-07-31 from https://ethereum.org/en/developers/

[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

```
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 2: 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.

[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] 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

[12] 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

[13] 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.

[14] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687* (2018).

[15] 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/