Psamathe: A Flow-Based DSL for Safe Blockchain Assets

Reed Oei[1], Michael Coblenz[2], and Jonathan Aldrich[3]

[1] University of Illinois, Urbana, IL, USA
reedoei2@illinois.edu
[2] University of Maryland, College Park, MD, USA
mcoblenz@umd.edu
[3] Carnegie Mellon University, Pittsburgh, PA, USA
jonathan.aldrich@cs.cmu.edu

1 Introduction

Blockchains are increasingly used as platforms for applications called *smart contracts* [16], which automatically manage transactions in an unbiased, mutually agreed-upon way. Commonly proposed and implemented applications include supply chain management, healthcare, *token contracts*, voting, crowdfunding, auctions, and more [10,9,8]. A *token contract* is a contract implementing a *token standard*, such as ERC-20 [1]. Token contracts are common on the Ethereum blockchain [17]—about 73% of high-activity contracts are token contracts [12]. Smart 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 [11].

Psamathe (/sɑmɑθi/) is a new programming language we are designing around *flows*, which are a new abstraction representing an atomic transfer operation. Together with *modifiers* and *locators*, flows provide a **concise** way to write smart contracts that **safely** manage assets, as explained in Section 2. Solidity, the most commonly-used smart contract language on the Ethereum blockchain [2], does not provide analogous support for managing assets.

A formalization of Psamathe is in progress [3], with an *executable semantics* implemented in the $\mathbb{K}$-framework [13], which is already capable of running the ERC-20 example shown in Figure 1.

Other newly-proposed blockchain languages include Flint, Move, Nomos, Obsidian, and Scilla [14,5,7,6,15]. 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 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, 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 *flows* describing the how values are transferred between variables. 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.

```
type Token is fungible asset uint256
transformer transfer(balances : any map ! address => any Token,
                     dst : ! address, amount : any 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.

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

- **Safer asset management**: In Psamathe, each flow is guaranteed to preserve the total amount of assets (except for flows that explicitly 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, 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 the necessary 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`, !, `nonempty`, or `every` ("!" means "exactly one"). 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 include `asset` and `consumable`. 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.

ERC-20 Comparison with 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 implementation in Figure 1 to Figure 2, which shows a Solidity implementation of the same function. 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.

```
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 [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.

3 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 an example smart contract 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 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. 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. Tokens (Aug 2020), https://github.com/ConsenSys/Tokens
5. 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)
6. 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)
7. Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. arXiv preprint arXiv:1902.06056 (2019)
8. 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
9. 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
10. IBM: Blockchain for supply chain (2019), https://www.ibm.com/blockchain/supply-chain/
11. 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
12. 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
13. 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
14. 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)
15. 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
16. 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
17. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)