LANGUAGE-NAME: A DSL for the Safe Management of Assets in Smart Contracts

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

ACM Reference Format:
Reed Oei. 2020. LANGUAGE-NAME: A DSL for the Safe Management of Assets in Smart Contracts. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

[How do authors/affiliations work for this?] LANGUAGE-NAME is a DSL for implementing programs which manage assets, targeted at writing smart contracts.

1.1 Contributions

We make the following contributions with LANGUAGE-NAME.

- **Flow abstraction**: LANGUAGE-NAME uses a new abstraction called a *flow* to encode semantic information about a program into the code. **[Is this a contribution or does it just enable the other contributions?]**
- **Safety guarantees**: LANGUAGE-NAME ensures that assets are properly managed, eliminating double-spend and asset-loss bugs.
- **Conciseness**: LANGUAGE-NAME makes writing typical smart contract programs more concise by handling common pitfalls automatically.

[Potential benefits of the language. Some of these are already discussed in the paper.

- **Good expression of financial assets: fungible, nonfungible/general uniqueness constraints, consumable vs. nonconsumable. NOTE: These are things that Obsidian doesn't express automatically. Emphasize the uniqueness stuff is actually not any more difficult/inefficient than existing solutions.**
- **Atomic flow construct encodes semantic intent**
- **Maybe the language is efficient, but would need an implementation to evaluate this.**
- **Flows are interesting?**

]

$\mathfrak{q}$	::=	$!$ | `any` | `nonempty`	(selector quantifiers)
$\mathcal{Q}$	::=	$\mathfrak{q}$ | `empty` | `every`	(type quantities)
T	::=	`bool` | `nat` | `map` $\tau \Rightarrow \sigma$ | t | ...	(base types)
τ	::=	$\mathcal{Q}\ T$	(types)
$\mathcal{V}$	::=	n | `true` | `false` | `emptyval` | ...	(values)
$\mathcal{L}$	::=	x | $x[x]$ | $x.x$	(locations)
E	::=	$\mathcal{V}$ | $\mathcal{L}$ | `total` t | ...	(expressions)
s	::=	$\mathcal{L}$ | `everything` | $\mathfrak{q}\ x : \tau$ **s.t.** E	(selector)
$\mathcal{S}$	::=	$\mathcal{L}$ | `new` t	(sources)
$\mathcal{D}$	::=	$\mathcal{L}$ | `consume`	(destinations)
F	::=	$\mathcal{S} \xrightarrow{s} \mathcal{D}$	(flows)
`Stmt`	::=	F | $S; S$ | ...	(statements)
M	::=	`fungible` | `nonfungible`	
	|	`consumable` | `asset`	(type modifiers)
`Decl`	::=	`type` t `is` $\overline{M}\ T$	(type declaration)
	|	`transaction` $m(\overline{x : \tau})$ `returns` $x : \tau$ **do** S	(transactions)
	|	...	
`Con`	::=	`contract` C { $\overline{\texttt{Decl}}$ }	(contracts)

Figure 1: A fragment of the abstract syntax of the core calculus of LANGUAGE-NAME.

2 LANGUAGE DESCRIPTION

2.1 Syntax

Figure 1 shows a fragment of the syntax of the core calculus of LANGUAGE-NAME, which uses A-normal form **[cite]** and makes several other simplifications to the surface LANGUAGE-NAME language. These simplifications are performed automatically by the compiler. **[TODO: We have formalized this core calculus (in K???).]**

2.2 Flows

The LANGUAGE-NAME language is built around the concept of a *flow*, an atomic, state-changing operation describing the transfer of a asset. Each flow has at least a *source* and a *destination*; they may optionally have a *selector* or a transformer. The source and destination are two storages which *provide* and *accept* assets, and the selector, if present, describes which part of the asset in the source should be transferred to the destination. If not present, all assets will be transferred.

All flows fail if the selected assets are not present in the source, or if the selected assets cannot be added to the destination. For example, a flow of fungible assets fails if there is not enough of the asset in the soruce, and a flow of a nonfungible asset fails if the selected value doesn't exist in the source location.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

There are two special kinds of assets: *fungible* and *nonfungible*. **[I think there's also assets which are neither fungible nor nonfungible.] [Not sure about these definitions.]** A *fungible* assets are those whose values are not unique and can be combined: for example, ERC-20 tokens are fungible, because two accounts may have the same number of tokens—the number isn't the token, but instead describes **how many** tokens there are. A *nonfungible* asset is an asset that is unique and immutable, and can be held in at most one location. For example, ERC-721 **[cite]** (discussed in more depth in Section **??**) tokens are nonfungible—each token is unique and can be held by at most one account at a time. LANGUAGE-NAME dynamically ensures that all newly created nonfungible assets are unique, and statically ensures that the resources are not duplicated or changed. **[This dynamic checking is no more costly that the standard approaches used for this purpose, should we discuss this?]** Furthermore, it supports data structures that make working with assets easier, such as *linkings*, which provide a useful abstraction of an "account" that holds a set of resources. **[Is it worth discussing linkings in more depth.?]**

3 CASE STUDIES

3.1 ERC-20

[Cite all Solidity code properly]

Figure 3 shows implementations of the ERC-20 **[cite]** standard in both Solidity and LANGUAGE-NAME, one of the most commonly implemented standards on the Ethereum blockchain **[cite]**. Only the core functions of `transfer`, `transferFrom`, and `approve` are shown, with the exception of `totalSupply` in the LANGUAGE-NAME implementation (included because to show the use of the **`total`** operator). All event code has been omitted, because LANGUAGE-NAME handles events in the same way as Solidity. This contract shows several advantages of the flow abstraction:

- **Precondition checking**: For a flow to succeed, the source must have enough assets and the destination must be capable of receiving the assets flowed. In this case, the balance of the sender must be greater than 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.
- **Data-flow tracking**: It is clear where the resources are flowing from the code itself, which may not be apparent in more complicated implementations, 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. This restriction prevents, in this example, tokens from being consumed, and can also be used to ensure that other assets, like ether, are not consumed.
- **Error messages**: When a flow fails, LANGUAGE-NAME provides **[TODO: **will provide**]** automatic, descriptive error messages, such as "`Cannot flow '<amount>' Token from account[<src>] to account[<dst>]: source only has <amount> Token.`". **[Not sure exactly what the error message should be.]** The default implementation provides no error message forcing developers to write their own. Flows enable the generation of the messages by encoding the semantic information of a **transfer** into the program, instead of using low-level incrementing and decrementing.

3.2 ERC-721

[Another benefit here is that linkings are a good datastructure for accounts of nonfungible assets. Of course, you could always implement a linking in a Solidity library... However, you still wouldn't have the property that only one account is guaranteed to hold each token, because of the nonfungibility/uniqueness.]

The ERC-721 standard **[cite]** requires many invariants hold: the tokens must be unique, at most one non-owning account can have "approval" for a token, we must be able to support "operators" who can manage all of the tokens of a user, among others. Because LANGUAGE-NAME is designed to handle assets, it has features to help developers ensure that these correctness properties hold. A LANGUAGE-NAME implementation has several benefits: because of the asset abstraction, we can be sure that token references will not be duplicated or lost; because `Token` has been declared as **`nonfungible`**, we can be sure that we will not mint two of the same token.

Figure **??** shows an implementation ERC-721's `transferFrom` function in both Solidity and LANGUAGE-NAME. The Solidity implementation is extracted from one of the reference implementations of ERC-721 given on its official Ethereum EIP page. In addition to the invariant required by the specification, there are also internal invariant which the contract must maintain, such as the connection between `idToOwner` and `ownerToNFTokenCount`, which are handled by LANGUAGE-NAME. This example demonstrates the benefits of having nonfungible assets and linkings built into the language itself.

3.3 Voting

[Solidity impl. comes from "Solidity by Example" page]

[Can include this section if we don't only want to talk about tokens...] The **`nonfungible`** modifier is useful in many programs, even those not dealing with financial assets, like ERC-721 contracts. We can also use **`nonfungible`** to remove certain incorrect behaviors from a voting contract, shown in Figure **??**.

3.4 The DAO attack

[Not sure how notable this is.] [Describe attack] We can prevent the DAO attack (the below is from https://consensys.github.io/smart-contract-best-practices/known_attacks/):

```
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    // At this point, the caller's code is executed, and
    // can call withdrawBalance again
    require(msg.sender.call.value(amountToWithdraw)(""));
    userBalances[msg.sender] = 0;
}
```

In LANGUAGE-NAME, we would write this as:

```
transaction withdrawBalance():
    userBalances[msg.sender] --> msg.sender.balance
```

Because of the additional information encoded in the flow construct, the compiler can output the safe version of the above code—reducing the balance before peforming the external call—without any user intervention.

```
contract EIP20 {
    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;
    function transferFrom(address from, address to, uint256 value)
        public returns (bool success) {
        require(balances[from] >= value &&
                allowed[from][msg.sender] >= value);
        balances[to] += value;
        balances[from] -= value;
        allowed[from][msg.sender] -= value;
        return true;
    }
    function approve(address spender, uint256 value)
        public returns (bool success) {
        allowed[msg.sender][spender] = value;
        return true;
    }
}
```

```
contract EIP20 {
    type Token is fungible asset uint256
    type Approval is fungible consumable asset uint256
    accounts : map address => Token
    allowances : map address => map address => Approval
    transaction transferFrom(src : address, dst : address, amount : uint256):
        allowances[src][dst] --[ amount ]-> consume
        account[src] --[ amount ]-> account[dst]
    transaction approve(dst : address, amount : uint256):
        allowances[msg.sender][dst] --> consume
        new Approval --[ amount ]-> allowances[msg.sender][dst]
}
```

Figure 2: A Solidity and a LANGUAGE-NAME implementation of the core functions of the ERC-20 standard.

4 DISCUSSION

5 RELATED WORK

[?]

6 CONCLUSION

```
contract NFToken {
  mapping (uint256 => address) idToOwner;
  mapping (uint256 => address) idToApproval;
  mapping (address => uint256) ownerToNFTokenCount;
  mapping (address => mapping (address => bool)) ownerToOperators;
  modifier canTransfer(uint256 _tokenId) {
    address tokenOwner = idToOwner[_tokenId];
    require (tokenOwner == msg.sender ||
             idToApproval[_tokenId] == msg.sender ||
             ownerToOperators[tokenOwner][msg.sender],
             NOT_OWNER_APPROWED_OR_OPERATOR);
    _;
  }
  modifier validNFToken(uint256 _tokenId) {
    require (idToOwner[_tokenId] != address (0), NOT_VALID_NFT);
    _;
  }
  function transferFrom(address _from, address _to, uint256 _tokenId)
    external override canTransfer(_tokenId) validNFToken(_tokenId) {
    require (idToOwner[_tokenId] == _from, NOT_OWNER);
    require (_to != address (0), ZERO_ADDRESS);
    address from = idToOwner[_tokenId];
    if (idToApproval[_tokenId] != address (0)) {
      delete idToApproval[_tokenId];
    }
    _removeNFToken(from, _tokenId);
    require (idToOwner[_tokenId] == _from, NOT_OWNER);
    ownerToNFTokenCount[_from] = ownerToNFTokenCount[_from] − 1;
    delete idToOwner[_tokenId];
    _addNFToken(_to, _tokenId);
    require (idToOwner[_tokenId] == address (0), NFT_ALREADY_EXISTS);
    idToOwner[_tokenId] = _to;
    ownerToNFTokenCount[_to] = ownerToNFTokenCount[_to].add(1);
}
```

```
contract NFToken {
    type Token is nonfungible asset nat
    type TokenApproval is nonfungible consumable asset nat
    balances : linking address <=> set Token
    approval : linking address <=> set TokenApproval
    ownerToOperators : linking address <=> {address}
    view canTransfer(_tokenId : nat) returns bool :=
        _tokenId in balances[msg.sender] or
        _tokenId in approval[msg.sender] or
        msg.sender in ownerToOperators[balances.ownerOf(_tokenId)]
    view validNFToken(_tokenId : nat) returns bool := balances.hasOwner(_tokenId)
    transaction transferFrom(_from : address, _to : address, _tokenId : nat):
        only when _to != 0x0 and canTransfer(_tokenId)
        if approval.hasOwner(_tokenId) {
            approval[approval.ownerOf(_tokenId)] −−[ _tokenId ]−> consume
        }
        balances[_from] −−[ _tokenId ]−> balances[_to]
}
```

Figure 3: A Solidity and a LANGUAGE-NAME implementation of the transferFrom function of the ERC-721 standard.

```
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;
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal(proposalNames[i], ));
        }
    }
    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;
    }
    function winningProposal() public view
            returns (uint winningProposal_) {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }
    function winnerName() public view
            returns (bytes32 winnerName_) {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

```
contract Ballot {
    type Voter is nonfungible asset address
    type ProposalName is nonfungible asset string
    chairperson : address
    voters : set Voter
    proposals : linking ProposalName <=> set Voter
    winningProposalName : string
    on create(proposalNames : set string ):
        chairperson := msg.sender
        new Voter(chairperson) --> voters
        new ProposalName --[ proposalNames ]-> (\name. name <=> {}) --> proposals
    transaction giveRightToVote(voter : address):
        only when msg.sender = chairperson
        new Voter(voter) --> voters
    transaction vote(proposal : string ):
        voters[msg.sender] --> proposals[proposal][msg.sender]
        if total proposals[proposal] > total proposals[winningProposalName] {
            winningProposalName := proposal
        }
    view winningProposal() returns string := winningProposalName
}
```

Figure 4: A Solidity and a LANGUAGE-NAME implementation of a simple voting contract.