OpenSC: A High-Level Programming Language Focusing on Smart Contract

Jun Sha js5506, Linghan Kong lk2811, Ruibin Ma rm3708, Rahul Sehrawat rs3688, Chong Hu ch3467

I. Motivation

OpenSC is an functional programming language which has similar functionality compared to Solidity. It is statically typed and will support several features. It is a high-level language that will be primarily used to implement smart contracts, which are programs that provide protocol for handling account behavior in Ethereum.

Compared to other languages, we will model contracts as some simple transition systems, with the transitions being pure functions of the contract state. These functions are expressed from one state to another state in a list of storage mutation, storage.

II. Language Paradigms and Features

OpenSC will be a high-level declarative language with strong, static typing, static scoping.

Beyond that, OpenSC will be a user-friendly language with high order functions, type inference, strict evaluation order and partial applications. Also, our language is focusing on Smart Contract and hence, we also provide built-in classes and functions to support that, such as Token, contract, log, event, storage.

TABLE I
Language Paradigms, Features and Built-in Functions

Paradigm	Declarative	**Features**	Type Inference High Order Functions Partial Applications
Type System	Strongly typed Statically typed	**Built-in Functions**	Message Token contract log event storage
Scoping	Statically Scoped	**Evaluation Order**	Strict

III. Hello World

```
/*
Below is an simple contract of having getter and setter function with return and guard parts.
    There are mainly four parts in define a method: guard which users can write pre and post
    condition, storage, effect which is event and emit in solidity and return.
*/
// : symbol means return should be followed by a type
signature Simple{
    //x is in storage
    storage x : UInt
    // two methods getter and setter
    method set: UInt
    method get: UInt -> UInt
    // define method set
```

```
    method public set(UInt x)  {
    guard{
        x >= 0 // since x is UInt
    }
    storage{
        data = x;
    }
    }
    // define method get
    method public get() : (UInt) {
    storage{
        return data;
    }
}
```

IV. OpenSC In One Slide

```
signature TOKEN{

  storage supply : UInt

  map balances : Address => UInt
  map allowances : (Address, Address) => UInt

  event Transfer = Transfer of (Address, Address, UInt)
  event Approval = Approval of (Address, Address, UInt)

  constructor c : UInt -> Unit
  method totalSupply : Unit -> UInt
  method balanceOf : Address -> UInt
  method transfer : (Address, UInt) -> Bool
  method transferFrom : (Address, Address, UInt) -> Bool
  method approve : (Address, UInt) -> Bool
  method allowance : (Address, Address) -> UInt


-- implementation

constructor c (s : UInt){
  storage{
    supply                |-> s
    balances[Env.sender]  |-> s
  }
  returns ()
}
method public totalSupply () : UInt{
  guard{
    Env.value == 0
    }
  returns supply
}

method public balanceOf (a : Address) : UInt{
  guard{
    Env.value == 0
    }
  returns balances[a]
}

method allowance (owner : Address, spender : Address){
  guard{
    Env.value == 0
    }
  returns allowances[(spender, owner)]
}

```

```
method transfer (a : Address, v : UInt) : bool{

  guard{

    Env.value == 0
    v =/= 0
    balances[Env.sender] >= v

    -- overflow checking
    (balances[Env.sender] - v) < balances[Env.sender]
    (balances[a] + v) > balances[a]
    }
  storage
    {
    balances[Env.sender] |-> balances[Env.sender] - v
    balances[a]          |-> balances[a] + v
    }
  effects{
    logs Transfer (Env.sender, a, v)
    }
  returns True
}

method approve (spender : Address, v : UInt) : bool{
  guard{
    Env.value == 0
    }
  storage{
    allowances[(spender, Env.sender)] |-> v
    }
  effects{
    logs Approval (Env.sender, spender, v)
    }
  returns True
}
method transferFrom (from : Address, to : Address, v : UInt)

  guard{

    Env.value == 0
    v =/= 0
    balances[from] >= v
    allowances[(Env.sender, from)] >= v

    -- overflow checking
    (allowances[(Env.sender, from)] - v) < allowances[(Env.sender, from)]
    (balances[from] - v) < balances[from]
    (balances[to] + v) > balances[to]
    }
  storage
    {
    allowances[(Env.sender, from)]  |-> allowances[(Env.sender, from)] - v
    balances[from]                  |-> balances[from] - v
    balances[to]                    |-> balances[to] + v
    }
  returns True

}
```