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. Introduction

`OpenSC` is a functional programming language which has similar functionality compared to `Scilla` [1] and `Pact` [2] . 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 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 mutations.

Inspired by the `MiniC` language, part of the `DeepSEA` compiler [3] , we aim to develop a language which allows interactive formal verification of smart contracts with security guarantees. From a specific input program, the compiler generates executable bytecode, as well as a model of the program that can be loaded into the `Coq` proof assistant. Our eventual goal is that smart contracts like storage, auction and token can be written by `OpenSC`, and that these contracts can be compiled via the translator into binary codes that can be executed on EVM.

We start from three basic types of simple smart contracts widely used in blockchain. `Simplestorage` is a simple storage contract program used to describe the process of storing data; `Auction` is an open auction contract program for people sending their bids where the auction is ended with the highest bid sent to the beneficiary; `Token` is a token implementation program to transfer tokens, as well as allow tokens to be approved.

II. Language Tutorial

`OpenSC` will be a high-level functional language for writing interface and methods. There are three main sections in `OpenSC`, signature, constructor, methods.

In signature section, there are storage and map declarations which are global variables user defined, event which may be emitted, constructor and method. In constructor section, user can initialize the declarations in interface with initial value and the constructor will always return void. Due to limit of our program, the constructor could not be generated as EVM bytecode. For methods section, user can define their own function for future use. In each function, there are four section is needed. Guard section is used to write the specification, storage section is used to show the changes of value in EVM storage; effects section is used to emit which we called log the event and returns section is used to return a value.

In signature section, users can define their declarations with type, identifier. Type supported in opensc signature section: boolean, int, uint, address, void and map. Moreover, user can define the identifier with letter and digit. The more details is in our language reference manual. Furthermore, there are five classes user allowed to define in the signature section, storage class, map class, event class, constructor class and method class. For example, in order to declare a storage data in signature, user need to first to declare storage class with an identifier and what type is that data. To be more specific, ”storage supply : UInt” is an example of declare a storage class with identifier, ”supply” and type uint. For map declaration, ”map id : (type list) => type” is the general format. For event declaration, ”event id = id of (type list)” is the general format. For constructor declaration, ”consturctor id : (type) -> void” is the general format. For method declaration, ”method id : (type list) -> type is the general format.

Although we do not support translate the constructor into evm bytecode for now, user can define a constructor with constructor name, parameter list, body and return type. For example,

```
constructor c (s : UInt){
  storage
    supply               |-> s;
    balances[Env.sender]  |-> s;
  returns void;
}
```

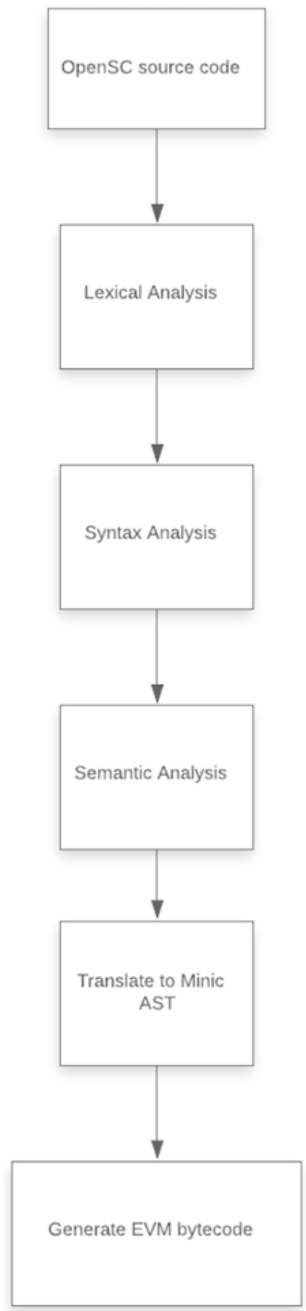
In order to implement a method in OpenSC, as mentioned, there are four main sections guard, storage, effects, returns except for function name and function body. Below is an example of transfer function in OpenSC:

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

  guard{
    Env.value == 0;
    balances[Env.sender] >= v;
    /- overflow checking -/
    balances[a] > balances[a] - v;
    balances[Env.sender] > balances[Env.sender] + v;
  }
  storage{
    balances[Env.sender] |-> balances[Env.sender] - v;
    balances[a]          |-> (balances[a] + v);
  }
  effects{

    logs Transfer (Env.sender, a, v);
  }
  returns True;
}
```

III. Architectural Design

Major components of the translator is shown in the following system block diagram.

Translator Architecture:

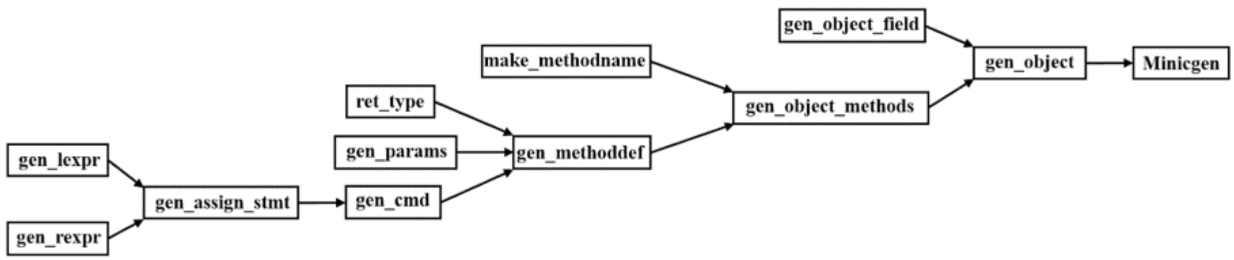


When the compiler takes the `OpenSC` source code as the input, it firstly does lexical analysis and syntax analysis using the scanner implemented in `scanner.mll` and the parser implemented in `parser.mly`, and generates a raw AST. Then, semantic analysis is done by the type checker implemented in `semant.ml` and a "semantically checked AST" will be generated.

Similar to the `minicgen.ml` IR generator in the `DeepSEA` compiler that we learned from, for our `OpenSC`, we also implemented a translator `translateMinic.ml` to translate our semantically checked AST into `MiniC` AST, which is the IR (intermediate representation). After that, the `MiniC` AST can be compiled into EVM bytecode using the backend which has already been implemented in the `DeepSEA` compiler. In such a way, a smart contract such as `simplestorage.sc` can finally be translated into EVM bytecode.

Specifically, key functions applied in the translation stage from our language SAST to `MiniC` AST are shown in the following flow chart.

Translate Details:



Also, there are some helper functions to connect and help implement each function. For example, backend_ident_of_globvar and backend_ident_of_tempvar are used to generate id with the help of function ident_generator. For some basic datatypes and operators, we have functions like gen_ctype, gen_unop, gen_binop as well as some type conversions to translate.

IV. Test Plan

We divide the test plan for four stages: parser, semantic check, Minic translation and bytecode compilation and test our program sequentially with four test files in Ocaml.

We provide 43 test cases in total to test the parser and semantic check. The parser can parse all test cases successfully and the semantic check step can pass the correct test cases and throw corresponding failure for different types of wrong test cases. Here are part of our test cases file names. The postfix _fail" and "_succ" represent whether this is a correct test case or a wrong test case; The prefix number \01_" represnent the test case id; the midfix describes what content this test case tests. From these test cases, we can show that our translator works well with the syntax we defined and checks different types, variables, functions correctly. We use Ocamlbuild to compile our two test files for parser and semantic check and run a bash script to do the automation test.

```
01_check_var_exist_succ.sc
02_check_var_exist_fail.sc
03_check_var_duplicate_announce_fail.sc
04_check_func_exist_succ.sc
05_check_func_exist_fail.sc
06_check_func_duplicate_announce_fail.sc
07_check_func_duplicate_implement_fail.sc
08_check_func_constructor_announce_once_fail.sc

...

40_check_map_query_wrong_type_fail.sc
41_check_map_query_key_type_not_allowed_fail.sc
42_check_map_query_assign_succ.sc
43_check_map_query_assign_unmatch_fail.sc
```

After our translator pass all the test cases, we write several more complex programs, translate them into Minic AST and compile them into EVM bytecode. We also write two `Ocaml` test file to test those two stages respectively. Notice that we need add tag "`-I backend`" and "`-pkg cryptokit`" during compiling our test file. Those source programs are more close to what we need in real smart contract. There are `Simple storage`, `auction` and `token`. We also list the output of Minic AST and bytecode to show our translator is functional. Notice that the backend of Minic is not supporting `Event` and `Log` function, the output is simply omitting those part. Here we only show simple storage and token due to page limitation. Those results show that our language `OpenSc` can be applied in reality and can translate from source code to ast, sast, Minic AST and eventually EVM bytecode.

- `simpleStorage`

```
/- A simple storage program -/

signature SimpleStorage {
    storage storedData : int;

  constructor c : (void) -> void;
  method set : (int) -> void;
}

constructor c (){
  storage
  returns void;
}

method set(x: int) {
  guard{
    x > 0 ;
  }
  storage{
      storedData |-> x;
  }
  effects{}
  returns voidlit;
}
```

Minic Ast:

```
---------------- VARS ----------------

* 1000100111

---------------- FUNCS ----------------



---------------- CONSTRUCTOR ----------------

empty

---------------- METHOD ZPOS11010010111101100111111001101010 ----------------

return type: void

params:
* 1000100110 : signed256

temps:

body:
* IFTHENELSE
* ├── BINOP
* │   ├── gt
* │   ├── TEMPVAR 1000100110 signed256
* │   ├── CONST_INT256 Z0 signed256
* │   ├── unsigned256
* ├── SKIP
* ├── REVERT
* ASSIGN
* ├── VAR 1000100111 signed256
* ├── TEMPVAR 1000100110 signed256
```

Bytecode:

```
5b5b38630000001560003938600f35b60006000fd5b7c0100000000000000000000000000000000000000000
00000000000000000600035048063d2f67e6a146300000003557600060000fd5b6004355b60008111630000004a
5763000000057565b806000555b905060006000f35b60006000fd
```

- `token`

```
/-
  token implementation satisfying the ERC20 standard:
   https://eips.ethereum.org/EIPS/eip-20

   interface
-/

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) -> void;
  method totalSupply : (void) -> 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 void;
}

method totalSupply (){
  guard{
    Env.value == 0;
  }
  storage{}
  effects{}
  returns supply;
}

method balanceOf (a : Address){
  guard{
    Env.value == 0;
  }
  storage{}
  effects{}
  returns balances[a];
  }

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

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

  guard{
    Env.value == 0;
    balances[Env.sender] >= v;
    /- overflow checking -/
    balances[a] > balances[a] - v;
    balances[Env.sender] > balances[Env.sender] + v;
  }
  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){

  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;
    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;
  }
  effects{}
  returns True;
}
```

Minic Ast: (Only show partial result for convenience)

```
--------------- METHOD ZPOS1100100111101000100010001010000 1 ---------------

return type: unsigned256

params:
* 1000101011 : unsigned256
* 1000101001 : unsigned256

temps:

body:
* IFTHENELSE
* ├── BINOP
* │   ├── eq
* │   ├── CALL0 callvalue void
* │   ├── CONST_INT256 Z0 signed256
* │   ├── signed256
* ├── SKIP
* ├── REVERT
* SKIP
* RETURN
* ├── HASHDEREF
* │   ├── VAR 1000101010 (unsigned256 unsigned256 hashmap)
* │   ├── TEMPVAR 1000101001 unsigned256
* │   ├── unsigned256

--------------- METHOD ZPOS1011010010000011101011111010011 ---------------
```

```
return type: unsigned256

params:
* 1000100111 : unsigned256
* 1000101100 : unsigned256

temps:

body:
* IFTHENELSE
* ├── BINOP
* │   ├── eq
* │   ├── CALL0 callvalue void
* │   ├── CONST_INT256 Z0 signed256
* │   ├── signed256
* ├── SKIP
* ├── REVERT
* IFTHENELSE
* ├── BINOP
* │   ├── ge
* │   ├── HASHDEREF
* │   │   ├── VAR 1000101000 (unsigned256 unsigned256 hashmap)
* │   │   ├── CALL0 caller void
* │   │   ├── unsigned256
* │   ├── TEMPVAR 1000101100 unsigned256
* │   ├── unsigned256
```

Bytecode:


```
5b5b386300000015600039386000f35b60006000fd5b7c010000000000000000000000000000000000000000
00000000000000006000350480637c80aa9f14630000036a578063e2e4263a146300000076578063c9e888a1
1463000000ea578063b483afd314630000012457806352192 09e1463000000ac5780630c41b0331463000002
0e5760006000fd5b6004355b60003414630000008b5763000000a6565b806001604052602052604060202054
5b9050600052602060 00f35b60006000fd5b6024356004355b600034146300000 0c4576300000 0e4565b8181
6002604052602052604060202055 5b60015b91505060005260206000f35b60006000fd5b6024356004355b60
003414630000010257630000011e565b816002604052602052604060202054 5b91505060005260206000f35b
60006000fd5b6024356004355b60003414630000013c5763000000208565b8133600160405260205260406020
20541015630000015b5763000000208565b818160016040526020526040602020540381600160405260205260
406020205411630000018957630000020 8565b81336001604052602052604060202054013360016040526020
5260406020205411630000 01b75763000000208565b81336001604052602052604060202054033360016040 52
6020526040602020555b8181600160405260205260406020205401816001604052602052604060202055 5b60
015b91505060005260206000f35b60006000fd5b6044356024356004355b600034146300000229576300000 3
64565b82816001604052602052604060202054101563000002485763000000364565b82336002604052602052
6040602020541015630000026757630000 00364565b3360026040526020526040602020548333600260405260
2052604060202054031063000002955763000000364565b806001604052602052604060202054838260016040
526020526040602020540310630000 02c3576300000 0364565b816001604052602052604060202054838360 01
604052602052604060202054011163000002f1576300000 0364565b82336002604052602052604060202054 03
3360026040526020526040602020555b82816001604052602052604060202054038160016040526020526040
602020555b82826001604052602052604060202054018260016040526020526040602020555b60015b925050
5060005260206000f35b60006000fd5b5b600034146300000 37c576300000 0389565b6000545b600052602060
00f35b60006000fd
```


V. SUMMARY

For the whole transalator, each component is implemented through cooperation of all the team members.

scanner: Linghan Kong, Chong Hu
parser: Linghan Kong, Chong Hu, Jun Sha
AST design: Linghan Kong, Ruibin Ma, Chong Hu
semantic analysis: Chong Hu, Linghan Kong, Ruibin Ma, Jun Sha
SAST design: Linghan Kong, Ruibin Ma, Chong Hu
translate MiniC: Ruibin Ma, Linghan Kong, Chong Hu, Jun sha

A. Linghan Kong

1) Summary: In this project, I have worked on different parts of our translator from the front end to back end. One thing that I am most impressive is that understand the whole progress of the translator is really important. At the first, I just focus on writing the front end without the understanding Minic properly and fully. Therefore, I need to revise the front-end of our translator after we begin writing the translator. Therefore, fully understanding how different parts works is one of the most important and what I learn a lot. Moreover, when reading the code, top down approach is a really good way to start with, when I first get the Deepsea compiler, I am not sure how to start to read the code. However, I use top-down approach to read the code and know each parts and how the minic ast looks like. Furthermore, communicating with different team members is also an important approach since exchanging ideas could be more helpful to understand the program.

2) Future Work: As a group, I think everyone did a great job and we share the knowledge base and what we learn in order to improve project. However, I would suggest people to start early the project and make a work pipeline.

B. Chong Hu

1) Summary: In this project, I'm mainly focusing on semantic check and part of parser and Minic translating. In the semantic check, I check our ast and map the ast to sast. In this processing, I spend a lot of time on revising sast and finding bugs repeatedly. In the later Minic translating, I have to do this again and again. This procedure is very boring. So, it is very important to have a better overview design and design ast and sast fitting what we need better. Beyond that, I realize that design and construct a language and corresponding translator need carefulness, patience and deep insight. If we can build a translator in the future, I believe we can do much better.

2) Future Work: All of my teammates did an excellent jobs in this language and I would like to thank every one of them. In such a huge team project, teamwork is one of the key to work efficiently and quickly. We need to package workload into different modules and distribute to every of us. If with appropriate arrangement in time schedule, we may not have such intensive workload in the end of semester.

C. Ruibin Ma

1) Summary: In this project, I worked on multiple parts with teammates such as sast.ml and translateMinic.ml. This was the first time I worked on a project using a functional language (OCaml), the first time I worked on a translator (front-end), and also the first time I collaborated with teammates remotely (due to COVID-19) most of time during the project. Therefore I have really learned a lot: First, functional programming is super cool; second, translator is interesting, and although it sounds complicated, there's no magic - as long as you spend enough time reading through the reference codes, in our case, the DeepSEA codes, and it's helpful to draw some flow charts; last but not least, (remote) pair (or triple, quad) programming is a more effective way than solo programming.

2) Future Work: I would suggest future teams not spending too much time on debugging alone but try to debug together, which is often much more efficient. If anyone is interested in continuing working on this project, preparing some basic background of blockchain and smart contracts and some working knowledge of OCaml or other functional programming languages could be helpful.

D. Jun Sha

1) Summary: In this project, I did quite a lot in different stages of our language compilation, especially the semantic analysis and minic translation. In the semantic analysis, I helped return a semantic-checked expression. For the minic translation, I have spent a lot of time understanding the details of translation frame, particularly what each function in the `minicgen.ml` means and how they are connected with each other. In our `translateMinic`, I implemented it from a top down perspective with my teammates, focusing on how the identlist can be generated and polishing some basic helper functions. Besides, in the early stage of our project, I added some pretty printing functions. Overall, I learnt a lot in this project, with a comprehensive understanding of how to design a new language and to translate it into a given IR. With this knowledge and experience, I have successfully got a compiler development intern in Alibaba.

2) Future Work: Since we need to polish some early parts when implementing other parts of the translator, it is important to start as early as possible. For example, while we are doing minic translation, we would modify and rewrite some parts of AST again and again. Also, if possible, it is helpful to regularly meet with the project advisor and get some useful suggestions from his perspective without taking a wrong path.

E. Rahul Sehrawat

My key takeaways from this course were the application of functional programming, understanding the translator, scanner and parser and learning the language of OCaml. I think the process of understanding a functional programming language such as OCaml is very useful in the workforce as more and more companies start to implement their use. Regarding the translator, scanner and parser, it was very cool to break apart grammer and understand the semantics and syntax like a computer would and then construct a robust language that a machine can process.

ACKNOWLEDGEMENTS

Thanks to Professor Ronghui Gu, the instructor of our course, who brought us to the PLT world and let us realize the charm of functional programming and formal verification, both of which are what our project is based on.

Thanks to River Dillon Keefer and Amanda Liu, TAs of our course, who introduced the DeepSEA project to us and provided very inspiring and helpful ideas on the OpenSC language syntax among other project details.

Thanks to Vilhelm Sjöberg, our project advisor, researcher at Yale and the primary creator of the DeepSEA project, who provided us with great information on everything about the DeepSEA project, and answered our many questions, which has been super helpful.

REFERENCES

[1] Language `Scilla`: https://scilla.readthedocs.io/en/latest/
[2] Language `pact`: https://github.com/kadena-io/pact
[3] Language `DeepSEA`: https://certik.io/blog/technology/an-introduction-to-deepsea