Verified Smart Contracts on Ethereum Blockchain

Overall Goal

Currently users of Ethereum write smart contracts in Solidity, which is compiled to EVM bytecode to be run on the blockchain. Any bugs in the contracts or the compiler can irreversibly lose people money. Our goal is to prevent these bugs by verifying the compiler's correctness and allowing properties of the contracts to be verified.

To verify the contracts, they will be rewritten in DeepSEA, a language which has a verified compiler to C/assembly and also generates Coq code. The associated Coq code can be extended with proofs about the functionality of the contract, for example proving that no money is lost from the system. Chris Fu's project is to research and write some of these proofs.

In order for DeepSEA contracts to be run on the Ethereum blockchain, we need to modify the compiler to C/assembly. This compiler will first transform the DeepSEA contract into a C-like abstract syntax tree, and then transform the C-like AST into EVM bytecode.

My project is to create the lower level of this compiler. First, I will modify the C-like intermediate language to better accommodate the data structures of Solidity and the memory model of the EVM. The next step would be to write the compiler from this intermediate language into EVM bytecode.

Details

Changes to Language Specification

The specification of the intermediate language, called Clight, is in Coq. The specification must be modified to add support for Solidity types and the EVM memory model. Here I've listed a few changes that I've predicted will be necessary.

- **Basic types:** Remove floating point numbers. Change the word-size from 32 bits (generic unix machine) to 256 bits (ethereum virtual machine). Cast between all sizes of integers from 8-256, in increments of 8. Deal with enums and contract addresses as integers.
- **Complicated types:** add variable-sized arrays and hash tables, and remove generic pointers. This complicates both rvalues and lvalues.
- **Machine state:** concept of global/persistent and local/transient variables to accompany the existing temps.
- **Expose properties of the EVM like gas usage and storage usage, to be used in proofs.
- **Small operations for logging, fetching hashes from blockchain, etc.**
Distinction between internal functions (a label in the contract’s code) and external functions (a function name and another contract’s address).

Compiler Implementation

In order to have anything to show for the project, I must write a compiler from modified-Clight to EVM bytecode.

It was proposed that this be written in OCaml, but I argue that it should be written in Coq. In order to be verified, it must eventually be translated into Coq anyway. I would like to shoulder any burden that Coq presents, such as proving that all recursion terminates. Furthermore, I find it easier to work in an all-Coq environment, without worrying about extracting OCaml.

Compilation Phases

Initially the compiler will output a list of EVM commands as a homemade datatype. These commands look similar to assembly language. For example it will use labels instead of direct address jumps.

Later in the project, I will integrate with the public formalization of the EVM, available here: https://github.com/pirapira/eth-isabelle.

This integration may involve writing an assembler from the list of EVM commands to an executable file.

List of some expected compilation phases (in approximate order)

1. Convert control flow statements like if-else and loops into more basic control flow patterns with labels and conditional jumps.
2. Convert complex operations like hash table lookups into more basic operations with addition and hashing. For rvalues this is wrapped in an Ederef, for lvalues the pointer is left bare.
3. Remove operations that don’t exist on the EVM, like << and !=, by converting to equivalent expressions.
4. Expression compilation: convert a tree of side-effect free expressions into a sequence of commands which produce the result value.
5. Statement compilation: convert a tree of statements into a sequence of EVM assembly commands, with jumps for navigation and side effects
6. Implement local functions and ethereum call contracts
7. Change bytecode that addresses stack locations as temp labels by using “register allocation” to allow temps to share stack positions. This is necessary because only the top 16 slots on the stack are addressable, and functions may want to use many more temps.
8. Change pushes of arbitrary integers to pushing constants of known size.
9. Convert jump labels into jumps to direct addresses in the contract code.
Solidity allows local variables, but DeepSEA doesn’t, so it wouldn’t make much sense to incorporate them into the compiler.

---

**Timeline**

Construct sample Clight function to test compiler at each stage
Generally, build up phases of compiler to handle more and more language constructs

- **Benchmark 2/7**
  - Make end-to-end example for payment channels.
  - Compiler works on easy expressions and statements.
- **Benchmark 2/21**
  - Decide how to change types, expressions, statements in Clight.
- **Benchmark 3/28**
  - Complex expressions and statements implemented.
- **Benchmark 4/11**
  - Compile public ethereum formalization and decide how to integrate it.
- **Benchmark 4/25**
  - Integrate ethereum formalization.
- **Benchmark 5/4**
  - Finishing touches.

---

**Deliverables**

At successful completion of this project I will have written the Coq code for a compiler from a C-like abstract syntax tree to an executable file that could be uploaded to the ethereum blockchain.

As a proof-of-concept, I will extract OCaml code for the compiler. Then write the Payment Channel from [https://github.com/amiller/sprites/blob/master/contractPay.sol](https://github.com/amiller/sprites/blob/master/contractPay.sol) in the C-like abstract syntax. Finally run the compiler on the abstract syntax and see if the resulting bytecode works.

---

**Extensions**

Break the compiler into layers designed to facilitate proving correctness.
Prove correctness of the compiler.
Generalize the DeepSEA compiler to Clight so my new backend can be swapped out for compcert, the certified C compiler to assembly.