Demo

The following are a few examples and explanations. I refer to the demo programs, which can be run with the instructions in the `README.md`.

Preliminaries

To summarize from my previous exposition, I built a K definition for a subset of the Solidity language (TinySol) and an associated meta-langauge for the virtual environment the langauge runs in. With the base K definitions, we can generate a parser and interpreter to execute TinySol. Semantics defined in the K-framework are touted as being able to generate all sorts of other tools for languages: compilers, model checkers, symbolic executors, deductive program verifiers, etc. TinySol itself is imperative and object-oriented, and the meta-language is for initializing state (balances, stored TinySol) and starting state transitions (transactions).

Demonstration of the generated parser and interpreter

Transaction semantics are as simple as Alice sending 5$ to Bob, or as complicated as any arbitrary program (programs should actually always halt but the feature for this didn't make it in). TinySol is the language for these arbitrary programs. Transactions can be distributed and concurrent, but their semantics in this model make it clear that their execution is simply sequential while the details of networked distributed transactions are abstracted away. That is, transactions in the network are collected in a node arbitrarily sequenced and executed one after the other. Invalid state updates (eg. insufficient balance) currently don't get completely reverted (no atomicity guarantee). Moreover, there is no true concurrency, but the stack based execution of the system can be exploited (more on this later).

Demo `tests/d1.k` demonstrates several transactions and their sequential execution. A transaction contains fields for the sender, recipient, value, and a function identifier to call with any necessary arguments.

A Wallet program

In `tests/d2.k` we have a virtual vallet that can only be spent by address 0. Address 0 must pay 1 unit for use. Indeed, address 0 uses the wallet to access a significant balance. The balances of all relevant addresses are first initialized, the contract is declared and instantiated, and transactions are sent. In the first transaction, address 0 successfully pays for use and transfers 150 units to address 2. The other two transactions are invalid since address 2 cannot spend the balance in the wallet and address 0 did not pay for use.

After execution, balances are updated accordingly (see the ledger cell) and the k-cell containing the original program has been rewritten to the unit value, as desired.

A more complicated example

`tests/d3.k` is of the so-called re-entrancy attack. It goes like this: suppose there is a contract A that can be called to buy event skins for a video game. Another contract B calls A to buy the skin. The contract checks if B has a high enough balance, and calls B's code delivery function to send the code, and finally deducts B's balance. B's delivery function could then call A's buy function again before A has

the chance to deduct B's balance. The transaction looks like it could be concurrent, but it is sequential since execution is concretely done through function call stack frames.

Indeed, the buyer contract only has enough balance (10 units) for one skin, but is able to re-enter the buy function of the seller before it can update the buyer's balance, resulting in all skins being lost (observe the location of the `skins` variable in the `objectClosure` in the `<store>` cell and its value at that location).