Rationale

For Educators and Recruters to trade securely, smart contract mechanism is required.

Requirements

The smart-contract should allow:

- Secure data trading with no trust to either side;
- Solving conflicts;
- Being extensible.

Accounts

There are 2 kinds of accounts:

- Personal: created for each client, directly belongs to that client;
- Robot - created by client, doesn't belong directly to that client (and anyone else); represents smart contract.

Robot account should contain, aside from token mapping:

- Data storage for smart contract state;
- The code to control the account, compiled to Plutus Core language.

The Plutus Core language allows declaring a module with exported (public) methods. These methods will be the public API for the account.

One can evaluate the Plutus Core code itself (not just call public methods) if account directly belongs to her. Personal accounts don't have any persistent associated code to control them.

Scripting

The Plutus language [1] will be used to program Robot nodes. Any interaction with account is done via Plutus code.

The evaluation cost is collected and summed into Fee.

To prevent account-multiplication/boolean-switch-scamming attacks, we must assign relatively high cost to such operations as:

- Robot-account creation;
- account "data" changes.

"Sandbox run" can be performed, to check costs and programming errors.

We will call natively-implemented functions to be called from Plutus Core "NIFs".

NIFs

NIF is an (ortogonal to each other) action to be invoked on Account. NIF is the only source of changes in Account.

The only operation outside the NIF scope is creation of account directly belonging to the client. That should be impossible to do with NIF for security reasons.

Any operation on the Account to be implemented should be represented as a call to NIF OR as a sequence of NIF-calls. This will allow us to reason about security/validity of operations with more confidence and limit the access and scope of each operation.

Each NIF has an invocation cost.

Transaction

We will cover "simple money transfer" and "data disclosure" transactions in this section.

Transferral transaction must contain:

- Sender signature;
- Receiver signature;
- Amount and kind of funds transferred (must be non-zero overall);
- "Nonce" to prevent double-invocation.
- (Optional) textual message for the receiver;
- (Optional) function call to be invoked upon transaction approval.

Transaction is the only way for accounts to interact.

Function call (if present) will contain the name of exported function and arguments to be supplied upon invocation.

The function would be invoked like that:

```
function-name(Seller-signature, amount, value1, value2, ..., valueN)
```

On successful code invocation, the money will be transmitted to the Target account and the costs will be demanded from the Sender. If code fails, the transaction is not published as successful and is rejected.

If there is not enough money supplied for the operation or the code raised an error, whole transaction will fail.

If there is no function call in transaction, the code invocation is assumed successful.

The fee for transaction approval will depend on NIFs invoked and will have cost calculated as `NifInvoked + C * sqr(NifInvoked)` where `C` is small enough so fast-ending transactions will not notice it; in the same time big transactions (like the one trying to evaluate `akkermann(10, 10)` will run out of gas).

Smart-contract mechanics

We assume that we have 2 sides:

- Buyer
- Seller.

"Gas" below is the estimation of the operation cost. The name and idea is taken from Etherium [2].

Smart contracts would work as follows. Buyer invokes a transaction which runs code directly on his account, that constructs a robot with following exported methods

- `initiate();`
- `accept-fee();`
- `accept();`
- `reject(block, proof);`
- `refuse();`
- `check-time(),`

carrying `Sum + Gas` amount of currency and some `predicate` to check the data.

Here is the state machine of that Robot-account:

```
(0) Account was created.
  "initiate" from Buyer:
    Send an invitation to buyer
    AND GOTO 1

  "reject" from Buyer: cleanup if something is wrong
    GOTO 4

(1) The trade was started.
      "accept" from Buyer
  AND "accept" from Seller: both accepted, terminating contract
    GOTO 3

  "reject" from Buyer
    GOTO 2

  "refuse" from Seller: in case smart contract doesn’t hold enough currency.
    GOTO 4

  "check-time" from Seller
    when time-spent > TIMEOUT
      GOTO 3

  "check-time" from Buyer
    when time-spent > TIMEOUT
    AND  Seller did not respond in time
      GOTO 4

(2) Arbitration.
  The robot invokes ‘check-block-and-proof‘ function.
  If it signals that Buyer is right (block invalidates the proof OR the predicate fails),
    GOTO 4
  else if Seller is right
    GOTO 3

(3)
  Sum is sent to Seller.
  GOTO 4.

(4) Cleanup. The account is closed and all remaining money
   are sent to the Buyer.
```

Example

Lets assume there are:

- Seller which has declared that he has his students’ Linear Algebra marks for Nov, 2018 worth 500 tokens (signed in some private Merkle tree);
- Buyer which has 600 tokens available.

We will consider three cases:

- Seller tried to send Buyer garbage instead of data;
- Buyer tried to blame Seller in giving her invalid data with data being completely valid (in terms of signature and predicate).
- Buyer decided to not `accept()` the contract.

All trades will have same initial part, so we will branch when nessessary. The trades will go as follows:

- Buyer formulates a predicate to check that data corresponds its description.
- She estimates the gas cost to be covered by 20 tokens max.
- Then she creates smart account using the scheme above with 500 + 20 tokens and the predicate.
- She checks that everything is right and invokes `initiate()`, which notifies Seller.
- Seller accepts and sends Buyer encrypted data via prvate channel, along with key.

  1. Seller tries to send garbage:
     - Buyer decrypts the data and finds that one block signature is invalid.
     - Then she invokes `reject(unencrypted-block, proof)` to start arbitration.
     - The smart account performs `check-block-and-proof(block, proof)` and finds that Buyer was right.
     - Then it returns all remaining funds to Buyer.
  2. Buyer tries to blame Seller with valid data:
     - Buyer selects the block to call "invalid".
     - Then she invokes `reject(unencrypted-block, proof)` to start arbitration.
     - The smart account performs `check-block-and-proof(block, proof)` and finds that Buyer was not right.
     - Then it send 500 tokens to Seller and all remaining currency to Buyer.
  3. Buyer receives the data, but remains silent:
     - Seller after some time calls `check-time()`.
     - If the timeout is expired, 500 tokens are sent to Seller.
     - All remaining currency is sent to Buyer.

References

[1] https://github.com/kframework/plutus-core-semantics/blob/master/docs/Plutus%20Core%20Grammar.pdf.

[2] http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html.