RIDE: a Smart Contract Language for Waves

A. Begicheva I. Smagin

May 30, 2018
v1.1

Abstract

The Waves Platform is a global public blockchain platform, providing functionality for implementing the most-needed scenarios for an account and token control. In this paper, we have presented our vision for Waves smart contracts as a two-level mechanism with a more detailed description of RIDE – the first-level language for smart contracts.

RIDE has smart accounts and smart assets for unloading the calculation system. This level has a simple-syntax functional language for scripting with pre-calculated complexity due to Turing-incompleteness. We have described the concept of RIDE and implementation of its structure and compilation process.

1 Introduction

In 1996, the computer scientist and cryptographer Nick Szabo first described smart contracts as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises" [1, 2]. Despite the development of technology since it was formulated, this definition is still accurate and captures the essence of a smart contract. The code of a smart contract should provide unconditional fulfillment of an established contract or set of rules and protect against incorrect actions, without recourse to intermediaries.

Smart contracts are traceable, transparent and irreversible, since they are hosted on the blockchain. Smart contracts must be guaranteed to complete, otherwise the network will fail. A smart contract language usually contains a signature verification mechanism.

Adding smart contracts brings the possibility of multi-signature mechanisms (multisig, escrow) or to withdraw funds according to certain conditions. Additionally, implementing smart contracts enables future use cases such as atomic swaps, oracles, multi-party lotteries or betting on sports games.

2 Existing Approaches

Bitcoin [3] includes a scripting system that is neither understandable nor expressive, with a Turing-incomplete zero-knowledge proof-based language. Due

to expressiveness limits, attempts to reuse this system are asymptotically more costly and time-consuming. Many of the tasks that can be solved using Bitcoin scripts, for example [6, 7], can be resolved more easily and efficiently if they are programmed in more understandable and less rudimentary language.

A fully Turing-complete language like Ethereum's Solidity [4] has some disadvantages: you cannot always determine the execution time of a contract written in such a language, and thus cannot determine the amount of gas needed to pay for the transaction. Ethereum pays miners certain fees that are proportional to the computational costs required by the smart contract. When a user sends a transaction to invoke a contract, the gas limit and the price for each gas unit must be specified. A miner who includes the transaction in a block subsequently receives the transaction fee corresponding to the amount of gas required for execution. If the execution of a contract requires more gas than the predefined limit, execution is terminated with an exception and the state is reverted to the initial state, but the gas is not returned. This is not ideal for users, because contracts have unpredictable complexity. Since smart contracts can transfer assets, besides correct execution it is also crucial that their implementation is secure against attacks aimed at stealing users funds. An analysis of possible attacks through smart contracts in Ethereum based on the limitations of its "gas" system has already been articulated [5].

3 Waves Approach

Our realisation of smart contracts will contain two parts: the first is a smart account language implementation and the second a foundational layer for developing various decentralised applications and smart contracts on the blockchain, with a built-in Turing-complete programming language. We see the syntax of our language as functional, similar to F#: strong and statically typed.

3.1 Smart Accounts

We plan to develop a more direct generalisation of Bitcoin scripting, a Turing-incomplete language which can still handle most of the use cases that can be undertaken by Turing complete languages.

A conventional account can only sign transactions before sending them to the blockchain. The idea of a smart account is the following: before the transaction is submitted for inclusion in the next block, the account checks if the transaction meets certain requirements, defined in a script. The script is attached to the account so the account can validate every transaction before confirming it.

The main requirement for our smart accounts is that they can be run for the price of normal transactions with a predefined fee, without any additional "gas" or other costs. This will be possible due to the statically predictable execution time. Since Waves has been built on top of an account-based model like Ethereum or Nxt (instead of Bitcoins input/output system), we can set security scripts on accounts.

In our vision, smart accounts cannot send transactions themselves or transfer funds according to given conditions, but can read data from the blockchain (for example, the height of a block or signatures from the transaction) and return the result of a predicate obtained on the basis of this data.

The language of smart accounts should be as simple as possible so that it is accessible to beginners or ordinary users who are not familiar with a particular language paradigm. Language grammar will be human-readable and user-friendly. We are consciously not going to provide users with the ability to write functions, recursions, and loops of indefinite nesting. We have explained earlier that we avoid constructions whose complexity cannot be predicted in advance and that cannot be executed in a definite number of steps. That is the reason why we also have no collections or for each constructions.

3.2 Smart Assets

If we plan to apply constraints on all operations for a specific asset, we cannot use a smart account. In our paradigm, we have smart assets for this purpose: the script will be attached to the asset and will work in a similar way. Transactions for such assets are valid only if the script returns True. For example, a script can verify proofs from a transaction, check if a notary/escrow approves the transaction, and that operations with the asset are not locked for a specified time. The script for the token is invoked upon the following operations with an asset:

- TransferTransaction
- MassTransferTransaction
- ReissueTransaction
- BurnTransaction

4 Use-cases

The main focus for the first version of smart accounts will be different security, integration, and crowdfunding cases.

An example of a security use case is multi-signature accounts. A multi-signature account is useful for contracts that need to be jointly owned, or shared, or when binding an agreement between multiple parties, or all of these. With its help, counterparties who do not trust each other can freeze a certain amount of tokens on the blockchain until the condition of having the required number of participants signatures is fulfilled.

The next group of use cases is integration, such as Oracles. An Oracle is the application that is responsible for connection to a given data source. It can place externally-sourced data on the blockchain as a series of transactions, but cannot change the data in them. Other people can receive money from a given account if this data meets the right conditions.

Conversely, if we want to remove a third party from an operation, a smart contract can be involved in the creation of an Atomic Swap - the next step in decentralisation. An Atomic Swap is a direct trade between two users of different cryptocurrencies, the honesty of which is guaranteed by a single contract in all relevant blockchains that cancels the transfer of funds back to the participants if the agreed exchange has not taken place. ("Atomic" in this definition means that an operation will either be performed completely, or it will not be executed at all.)

Crowd sale processes like selling tokens on an exchange can be implemented trustlessly on Waves DEX without smart contracts. However, smart accounts can help investors after an ICO. For example, they can be used to control fund use via escrow, token holder voting, etc.

5 Implementation

Any expression in our concept is a simple typed expression tree without cycles. The complexity of such an expression is calculated with a maximum complexity of tree branches. Therefore a smart account code requires a statically-predictable amount of resources for execution, such as time, memory or CPU.

While the user writes smart account code in a high-level language, the Waves Contracts execution engine is a straightforward evaluator of a low-level expression tree within context. In order to achieve that, there are several stages which make text script produce an execution result.

The first stage is parsing. The Parser builds an untyped abstract syntax tree (AST) from script text. Only syntax rules are checked at this phase, like correct variable names, function invocation with () and so on.

The second stage is type checking and compiling. In this stage the untyped AST is enriched with types and types are checked, according to function signatures. It operates within a context of type definitions, types of defined values and predefined function signatures. An expression operates `BLOCK`, which consists of `EXPRs`. Each `EXPR` has a type and is one of:

- `LET(name, block)` to define a variable
- `GETTER(expr, fieldName, resultTtype)` to access field of structure
- `FUNCTION_CALL(name, argBlocks, resultType)` to invoke a predefined function within context
- `IF(clause, ifTrueBlock, ifFalseBlock, resultType)` for lazy branching
- leafs: `CONST_LONG(long)`, `CONST_BYTEVECTOR(byteVector)`, `CONST_STRING(string)`, `REF(name, resultType)`

Table 1: Predefined Functions Smart Account language

check if the option contains some value	`isDefined`
extract the value from the option	`extract`
create the option	`some`
return size of byte array	`size`
return transaction by ID	`getTransactionById`
get long value from DataTransaction by key	`getLong`
get boolean value from DataTransaction by key	`getBoolean`
get byte array from DataTransaction by key	`getByteArray`
get address from public key	`addressFromPublicKey`
get address from string	`addressFromString`

This set does not include constructs that are used exclusively for ease of parsing. This step is important to validate user input and the output of this stage is exactly what is sent to the blockchain.

The third stage is the evaluator, which operates a typed expression tree within a context. It traverses the low-level typed AST, produced at the previous step, returning either the execution result or an execution error. The *Context* contains a map of predefined functions with implementation, predefined types and lazy values that can be calculated upon calls within the given tree path.

The high-level Smart account code is a logic formula that combines predicates over a context (blockchain state and transaction) and cryptographic statements (Table 2) and functions from Table 1. For standard actions the binary operations: $>=, >, <, <=, +, -, \&\&, ||, ==$ and unary $-, !$ are available. Lazy constants declaration are implemented via the `let` keyword, as in the F# language. There is an IF-THEN-ELSE clause, and access to fields of any instances of predefined structures is implemented via . (e.g. someInstance.feldOne). Calls to predefined functions is implemented via (). Access to the element of a List is performed using [].

It is an important property that the smart account does not store any data on the blockchain. A smart account will only have access to blockchain state values that can be retrieved and executed relatively fast, in a "constant" time, for example to such fields as:

- balances of accounts;
- access to account state;
- current Block's properties (e.g. height and timestamp);
- data stored in other transactions referenced by transactions (e.g. proofs, DataTransaction).

The types which will be used to predicate are `LONG`, `BOOLEAN`, `STRING`, `Option[T]`, `byteVector`, `List`, `Nothing`. An `Option[T]` can be either `Some[T]`

Table 2: Cryptographic functions in Smart Account language

Operator name	Syntax	Parameters
signature validation	`sigVerify(sign)`	`sign` - signature from transaction, we use elliptic curve signature, version curve25519
hash computation for keccak256	`keccak256(message)`	`message` - byte array for hashing
hash computation for blake2b256	`blake2b256(message)`	`message` - byte array for hashing
hash computation for sha256	`sha256(message)`	`message` - byte array for hashing

or `None` object, which represents a missing value. For example, if we want to get some transactions that do not exist in the blockchain we should receive `None`, but if this transaction is contained in the blockchain, the method should return `Some(transaction)`. A user cannot create new types; only predefined ones are available. `DataTransaction` can set/overwrite a typed primitive value for a key on account of sender.

All constants will be declared in lazy `let` constructions, which delays the evaluation of an expression until its value is needed, and does it at most once. For instance:

```
let hash = blake2b156(preImage).
```

The `hash` is not a variable: once created its values never change.

`SetScriptTransaction` sets the script which verifies all outgoing transactions. The set script can be changed by another `SetScriptTransaction` call unless it's prohibited by a previously set script.

6 Examples

6.1 Multi-Signature Account

Suppose that there are 3 people in a team and they hold common funds for corporate purposes. It is convenient for the team to make a decision about the allocation of common funds according to the majority decision, and they use a multi-signature account to do this. They create an account and do

`SetScriptTransaction` with the multi-sig account, which can be implemented as follows:

```
let alicePubKey  = base58'B1Yz7fH1bJ2gVDjyJnuyKNTdMFARkKEpV'
let bobPubKey    = base58'7hghYeWtiekfebgAcuCg9ai2NXbRreNzc'
let carolPubKey = base58'BVqYXrapgJP9atQccdBPAgJPwHDKkh6A8'

let aliceSigned = if(sigVerify(tx.bodyBytes, tx.proof[0], alicePubKey))
    then 1 else 0
let bobSigned = if(sigVerify(tx.bodyBytes, tx.proof[1], bobPubKey))
    then 1 else 0
let carolSigned = if(sigVerify(tx.bodyBytes, tx.proof[2], carolPubKey))
    then 1 else 0

aliceSigned + bobSigned + carolSigned >= 2
```

Here users gather 3 public keys in `proof[0]`, `proof[1]` and `proof[2]`. The account is funded by the team members and after that, when at least 2 of 3 team members decide to spend money, they provide their signatures in a single transaction. The Smart account script validates these signatures with proofs and if 2 of 3 are valid then the transaction is valid too, or else the transaction does not pass to the blockchain. Note that after the `SetScriptTransaction` operation all non multi-signature transactions are discarded.

6.2 Smart Asset With Notary Proof

Suppose that we want to transfer some assets only when we receive proof of possession from the notary, who we can trust. Also, we want to check that the recipient is able to accept the transfer. For this purposes we can write a script for our smart asset:

```
let king = extract(addressFromString("${king.address}"))
let company = extract(addressFromString("${company.address}"))
let notary1 = addressFromPublicKey(extract(getByteArray(king,"notary1PK")))
let txIdBase58String = toBase58String(tx.id)
let notary1Agreement = getBoolean(notary1,txIdBase58String)
let isNotary1Agreed = if(isDefined(notary1Agreement))
    then extract(notary1Agreement)
    else false
let recipientAddress = addressFromRecipient(tx.recipient)
let recipientAgreement = getBoolean(recipientAddress,txIdBase58String)
let isRecipientAgreed = if(isDefined(recipientAgreement))
    then extract(recipientAgreement)
    else false
let senderAddress = addressFromPublicKey(tx.senderPk)
senderAddress.bytes == company.bytes ||
    (isNotary1Agreed && isRecipientAgreed)
```

In `isNotary1Agreed` there is a check to confirm that the transfer transaction has the signature of the notary. In `recipientAgreement` we check that the recipient allows the current transaction, and in the end we return `true` if both of these conditions are satisfied, or `false` otherwise.

For example transactions list will be:

1. kingDataTransaction → DataTransaction for king's address with BinaryDataEntry("notary1PK", ByteStr(notary.publicKey))
2. transferFromCompanyToA → TransferTransactionV1 from accountA to accountB
3. notaryDataTransaction → DataTransaction for notary's address with BooleanDataEntry(transferFromAToB.id().base58, true)
4. accountBDataTransaction → DataTransaction for accountB's address with BooleanDataEntry(transferFromAToB.id().base58, true)
5. transferFromAToB → TransferTransactionV1 from accountA to accountB

7 Summary

Smart contracts are an important mechanism for any blockchain platform and their realisation should be convenient and understandable for people. In this paper we have presented our vision for Waves smart contracts as a two-level mechanism.

The first level has smart accounts and smart assets for unloading the calculation system. This level has a simple-syntax functional language for scripting and all scripting on this level has pre-calculated complexity due to Turing-incompleteness. This approach covers critical requirements for smart contracts and also provides a good basis for further development of a fully-fledged Turing-Complete second level to our smart contracts.

The second level will allow the creation of decentralised applications on the blockchain, which will be able to send transactions themselves.

References

[1] Nick Szabo *Smart Contracts: Building Blocks for Digital Markets.* EXTROPY: The Journal of Transhumanist Thought,(16), 1996.

[2] Nick Szabo. *The idea of smart contracts.* http://szabo.best.vwh.net/smart contracts idea.html, 1997.

[3] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system.* bitcoin.org, 2009.

[4] Ethereum Foundation. *Ethereums white paper* https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[5] Atzei, Nicola and Bartoletti, Massimo and Cimoli, Tiziana. *A survey of attacks on Ethereum smart contracts (SoK)*. International Conference on Principles of Security and Trust, 164–186, Springer, 2017.

[6] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. *Secure Multiparty Computations on Bitcoin*. IEEE Symposium on Security and Privacy, 2013.

[7] Rafael Pass and Abhi Shelat. *Micropayments for decentralized currencies*. Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS 15, 207-218, 2015.