MOGRA: a Smart Contract Language for Waves

A. Begicheva I. Smagin

September 4, 2018
v1.2

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 present our vision for Waves smart contracts as a two-level mechanism with a more detailed description of MOGRA – the first-level language for smart contracts.

MOGRA has smart accounts and smart assets for embed the calculation system. This level has a simple-syntax functional language for scripting with pre-calculated complexity due to Turing-incompleteness. We describe the concept of MOGRA 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 realization of smart contracts has two parts: the first is a smart account language implementation and the second a foundational layer for developing various decentralized 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 developed a more direct generalization 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 is 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 Bitcoin's 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 is 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 'for each' constructions.

4 Use-cases

The main focus for the first version of smart accounts is 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 an 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 decentralization. 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 trustlessly implemented 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

5.1 Phases

In the implementation of our smart contract language we have 5 stages:

1. parsing,
2. compilation,
3. deserialization,
4. cost computation,
5. evaluation.

The first two stages are off-chain stages and we will not pay attention to them in this article. The deserialization, the cost computation, and the evaluation are on-chain stages with special implementation and they will be discussed below.

Any expression in our concept is a simple expression tree without cycles. The complexity of such expressions 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 memory or CPU.

While the user writes a 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 third stage is deserialization. In this stage MOGRA only checks syntax rules, like correct variable names, function invocation with () and so on.

The forth stage is a cost calculation. It operates within a context of type definitions, types of defined values and predefined function signatures. An expression operates the base type `EXPR`, and its sub-type `BLOCK`. Each `EXPR` has a type and is one of:

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

The last stage is the evaluator, which operates an expression tree within a context. It traverses the low-level 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, user's functions, predefined types and lazy values that can be calculated upon calls within the given tree path.

5.2 Language Description

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 are implemented via (). Access to elements of a List is performed using [].

MOGRA has no cycle and recursion possibility, unlike Solidity. We should note that MOGRA as a language is not Turing-complete due to the lack of the possibility of creating loops or any other jump-like constructions. At the same time, it can be Turing-complete when used in conjunction with a blockchain, since theoretically the blockchain has an infinite length and we have other possibilities, e.g. DataTransaction. This kind of transaction provides data for smart contracts to work with. For example, if an oracle publishes some data once in a while using a publicly known account, smart contracts can use that data in their logic. In this article [10] it is shown that Turing-completeness of a blockchain system can be achieved through unwinding the recursive calls between multiple transactions and blocks instead of using a single one, and it is not necessary to have loops and recursion in the language itself.

All constants are 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 = blake2b256(preImage).
```

The `hash` is not a variable: once created its values never change, and all structures are immutable.

`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.

There is a mechanism for checking a value against a pattern and you can handle the different expected types in a match expression. A match expression has a value, the match keyword, and at least one case clause:

```
match tx {
case t:Transfer => t.recepient
case t:MassTransfer => t.transfers
case _ => throw()
}
```

`throw()` signals the occurrence of an exception during a script execution. In case of throw the transaction does not pass into the blockchain.

5.3 Standard Library

It is an important property that the smart account does not store any data in the blockchain. A smart account has access to blockchain state values that can

Table 1: Waves context functions

Operator name	Syntax	Description
`transactionById`	ByteVector ⇒ Option[Transaction]	return a transaction by ID
`transactionHeightById`	ByteVector ⇒ UNION(LONG, UNIT)	return a transaction height by ID
`addressFromRecipient`	Option(ByteVector) ⇒ addressType	get an address from recipient
`assetBalance`	addressOrAliasType ⇒ Long	provide balance info for any account
`addressFromPublicKey`	ByteVector ⇒ addressType	return an address by public key
`addressFromString`	String ⇒ UNION(addressType.typeRef, UNIT)	return an address by string
`wavesBalance`	addressOrAliasType ⇒ Long	return waves balance by address
`toBytes`	value: Boolean | Long | String ⇒ ByteVector	return a bytes of boolean/long/string value
`toString`	value: Boolean | Long ⇒ String	return a string from boolean/long

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 high-level smart account code is a logic formula that combines predicates over a context (blockchain state and transaction) and cryptographic statements (Table 3) and functions from Table 1 and Table 2.

The DataTransaction can set/overwrite a typed primitive value for by an address of sender as a key. The fields can be accessed from WavesContracts by key.

The types which are used to predicate are `Long`, `Boolean`, `String`, `byteVector`, `List[T]`, `Nothing`, `Unit`, `UnionType`. `UnionType` can be combination of many types e.g. `UnionType(type*)` or be an object, which represents a missing value `None`. A user cannot create new types; only predefined ones are available.

Table 2: Waves context functions for DataTransaction

Operator name	**Syntax**	**Description**
`getInteger`	(accountAddress: ByteVector, key: String) ⇒ Option[Long]	get a long value from DataTransaction or from state by key
`getBoolean`	(accountAddress: ByteVector, key: String) ⇒ Option[Boolean]	get a boolean value from DataTransaction or from state by key
`getBinary`	(accountAddress: ByteVector, key: String) ⇒ Option[ByteVector]	get a byte vector from DataTransaction or from state by key
`getString`	(accountAddress: ByteVector, key: String) ⇒ Option[String]	get a string value from DataTransaction or from state by key
`getInteger`	(accountAddress: ByteVector, index: Long) ⇒ Option[Long]	get a long value from DataTransaction array by index
`getBoolean`	(accountAddress: ByteVector, index: Long) ⇒ Option[Boolean]	get a boolean value from DataTransaction array by index
`getBinary`	(accountAddress: ByteVector, index: Long) ⇒ Option[ByteVector]	get a byte vector from DataTransaction array by index
`getString`	(accountAddress: ByteVector, index: Long) ⇒ Option[String]	get a string value from DataTransaction array by index

Table 3: Cryptographic functions in Smart Account language

Operator name	Syntax	Description
`sigVerify(body, sign, pubKey)`	(body: ByteVector, signature: ByteVector, pubKey: ByteVector) ⇒ Boolean	signature validation
`keccak256(message)`	ByteVector ⇒ ByteVector	hash computation for keccak256
`blake2b256(message)`	ByteVector ⇒ ByteVector	hash computation for blake2b256
`sha256(message)`	ByteVector ⇒ ByteVector	hash computation for blake2b256
`toBase58String'`	ByteVector ⇒ String	converting to Base58
`toBase64String'`	ByteVector ⇒ String	converting to Base64
`fromBase58String'`	String ⇒ ByteVector	converting from Base58
`fromBase64String'`	String ⇒ ByteVector	converting from Base64

5.4 Halting Problem

5.4.1 Termination of Deserialization of Contract

The deserialization stage builds an abstract syntax tree (AST), a directed acyclic graph, from script text. A complete description of such grammar is a set of rules that determines all non-terminal symbols of the tree so that each non-terminal symbol can be reduced to a combination of a terminal symbol (leaf) by successive application of the rules. As a top-down parsing strategy, it always halts when all expression converts into terminal symbols.

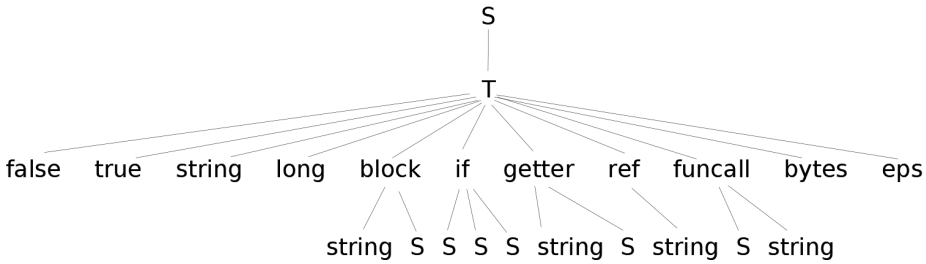

Figure 1: Deserialization tree for MOGRA bytecode

The formal representation of bytecode language:

$$
\begin{aligned}
&S \rightarrow T \\
&T \rightarrow long \\
&T \rightarrow string \\
&T \rightarrow bytes \\
&T \rightarrow true \\
&T \rightarrow false \\
&T \rightarrow if\ S\ S\ S \\
&T \rightarrow block \\
&block \rightarrow string\ S\ S \\
&T \rightarrow ref \\
&ref \rightarrow string \\
&T \rightarrow getter\ string \\
&getter \rightarrow string\ S \\
&T \rightarrow string\ S \\
&T \rightarrow funcall\ S \\
&S \rightarrow \epsilon \\
&funcall \rightarrow string \mid long\ S^*
\end{aligned}
$$

Note that the `long, string, bytes, true, false` are terminals and `eps` is ϵ. A full deserialization tree for this grammar is presented in Figure 1. Each leaf of a tree is either a terminal or leads to right-recursion.

Let us prove that the parser always halts. As an input we have a byte array, that is always finite, based on the fact that the user cannot write an infinite amount of code. The parser processes all incoming bytes in turn, from the left to the right in accordance with the described grammar. The parser is a top-down one, begins with the start symbol and systematically applies the rules of the CFG (context-free grammar) until there are no more non-terminal symbols left. We have no left-recursion: all recursions are finite, since we have a finite set of bytes and always move the pointer to the right and never to the left. The pointer is moved on every step, and if there is an error, the parser stops.

Bytecode consists of several parts. The first byte is always a type byte (for 1 from 10 types). Then a structure can be different and depends on type, which can be seen in Figure 2 and Figure 3.

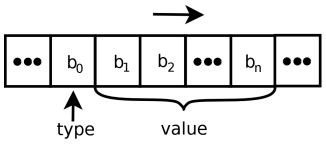

Figure 2: The structure for terminal types `long, true, false`

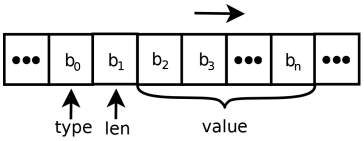

Figure 3: The structure for terminal types `string, bytes`

All terminal functions from grammar in the parser code consists of `get` operations from `java.nio.ByteBuffer` (see documentation [11]), that always move to the next byte in the buffer. All elements in the buffer start with the special byte of type and then continues to record the value's bytes or length in bytes and value bytes.

5.4.2 Termination of cost calculation stage

The cost computation stage is important to validate the cost of user input and the output of this stage is exactly what is sent to the blockchain. The analyzed expression is a finite set of tree leaves, from AST that is built in the first step, so this stage also always halts, since the tree has a finite structure.

Let us consider the procedure for calculating the cost of the next example:

$$
\begin{aligned}
&let \;\; x \;=\; y \\
&let \;\; y \;=\; x \\
&x \;+\; y
\end{aligned}
$$

In this fictional example, that will raise an error at the evaluation stage, we define `x` as `y` and vice versa, and we show that it does not get stuck in a cycle. Our lazy computation approximator calculates variables with `let` only when this variable appears in some expression. Thus, for this example, the approximator calculates `x` and `y` when it encounters in `x + y`. It puts variables in the special array `Arr` and removes them from it after calculating the complexity of the corresponding `let`. So, we put `x` and `y` into `Arr`. Then we remove the `x` from the array and calculate its complexity as `ref + complexity(y)`, where `ref` is a complexity of reference. Then we calculate `y` complexity as `ref` and the total is equal to `ref + ref + sum + ref`, where `sum` is complexity of addition.

We add and remove each variable only once, as we can see from the example above. As soon as there are no elements left in the `Arr` array and no tree leaves without an assessment of complexity, the cost computation stage is completed.

5.4.3 Termination of evaluation stage

The third stage is an evaluation, which operates an expression tree within a context. It traverses the low-level AST, produced at the previous step, returning either the execution result or an execution error.

Let's prove the fact that the evaluation stage always halts by using structural induction on the AST.

Claim: If the root of the AST is visited once, then each node in the in AST is visited at most once.

Proof:

Base: Suppose that the root is of a leaf type `(CONST_*, TRUE, FALSE)`: the root is the only node of the AST, so the statement is trivially true.

Induction:

- `LET`s and `REF`s case: Suppose the root is of type `BLOCK`. Then it has two children: `INNER` and `LET`. The same reasoning as for `FUNCTION_CALL` above applies to `INNER`. The left side of the `LET` has no children except the name, which is visited once. As far as the expression on the right side of the `LET` goes, it is not visited. Rather, a new pointer to it is created in the context map. However, it can be visited at most once from the context map, because once it is visited via REF, it is replaced with its value and not present in the map again. Therefore, by inductive hypothesis, the same applies to its children.

- No `LET`s-`REF`s case: Suppose the root is of type `FUNCTION_CALL`, `GETTER`, or `IF`. The corresponding eval function (`evalFunctionCall`, `evalGetter`, and `evalIF`) visits each child at most once. There is no other way to visit the child of the root except to do it from the root, because children of the root have only one pointer to them – namely, the root (by definition of a tree). Therefore, by inductive application of the claim, each node in every subtree of the root will be visited at most once.

Since upon contract execution, the expression root is visited once and, as we have shown above, all its descendants are visited only once, the whole computation halts.

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 cooperPubKey = base58'BVqYXrapgJP9atQccdBPAgJPwHDKkh6A8'

let aliceSigned  = if(sigVerify(tx.bodyBytes, tx.proofs[0],
                   alicePubKey)) then 1 else 0
let bobSigned    = if(sigVerify(tx.bodyBytes, tx.proofs[1],
                   bobPubKey)) then 1 else 0
let cooperSigned = if(sigVerify(tx.bodyBytes, tx.proofs[2],
                   cooperPubKey)) then 1 else 0

aliceSigned + bobSigned + cooperSigned >= 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 Atomic Swap

An atomic swap is the exchange of one cryptocurrency for another without the participation of third parties. In the MOGRA language an atomic swap can be written as:

```
let Bob = Address(base58'$BobBC1')
let Alice = Address(base58'$AliceBC1')
match tx {
  case ttx: TransferTransaction =>
     let txToBob = (ttx.recipient == Bob) && (sha256(ttx.proofs[0])
                == base58'$shaSecret') && ((20 + $beforeHeight)
                >= height)
     let backToAliceAfterHeight = ((height >= (21 + $beforeHeight))
                && (ttx.recipient == Alice))
     txToBob || backToAliceAfterHeight
  case other => false
}
```

where `$shaSecret` is `sha256` of "some secret message from Alice", and `$beforeHeight` is some predefined height.

For example, transaction's list will be:

1. `TransferTransactionV2` from `AliceBC1` to `swapBC1`
2. `TransferTransactionV2` from `swapBC1` to `BobBC1` OR after some height from `swapBC1` to `AliceBC1`

7 Summary

Smart contracts are an important mechanism for any blockchain platform and their realization 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 embed 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 decentralized 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. *Ethereum's 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.

[8] Fu, King Sun. *Syntactic methods in pattern recognition*. Elsevier, vol.112, 1974.

[9] Hartson, H Rex and Hix, Deborah *Advances in human-computer interaction*, Intellect Books, vol.2, 1988.

[10] Chepurnoy A., Kharin V., Meshkov D. *Self-Reproducing Coins as Universal Turing Machine* arXiv:1806.10116. 2018.

[11] Oracle docs: java.nio.ByteBuffer, https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html, 2011.