Ride: a Smart Contract Language for Waves

A. Begicheva I. Smagin

September 25, 2019
v2.0

Abstract

The Waves Platform is a global public blockchain platform, providing functionality for implementing the most-needed scenarios for account and token control. In this paper, we present our vision for Waves smart contracts with a more detailed description of Ride – the language for smart contracts.

Ride has smart accounts and smart assets to embed the calculation system. It is a simple-syntax functional language for scripting, with pre-calculated complexity due to Turing-incompleteness. We describe the concept of Ride and implementation of its structure and compilation process.

Ride allows creation of decentralized applications (dApps), which can be invoked by other accounts. Such invocation may result in state changes. We describe dApps implementation and discuss the opportunities they open up.

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. The completion of smart contracts must be guaranteed, 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 the ability to withdraw funds according to certain conditions. Additionally, implementing smart contracts enables the future development of dApps – decentralized applications that can be built not only for financial use cases, but for any purpose, for example for gaming or voting.

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 [5, 6], 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] was designed to be "familiar" for the widest range of developers possible, therefore inheriting a lot of bad practices that modern languages try to avoid: with mutable data structures it is easier to miss a potentially costly error, and it is harder to formally verify code written in an imperative language.

Ethereum pays miners certain fees that are proportional to the computational costs required to execute 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 may 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 [7].

3 Waves Approach

3.1 About Turing Completeness

Blockchain architecture is not the best environment for conducting computations. The reason is the very nature of the blockchain's approach: replication of everything by as many independent parties as possible. For example, Ethereum's Solidity is Turing-complete and promoted as such, but the system must be restricted artificially with gas in order to remain invulnerable to falling into an infinite loop or another similar problem. The system should be able to achieve the same functionality as in Turing-complete systems, without introducing any gas and other artificial restrictions. Ethereum's gas appears to be the only solution for these problems in a Turing-complete language. However, the result of such an approach is that failed transactions that have been paid for will stay on a blockchain and be replicated by many nodes, which thereby repeat the same work over and over without providing any value.

3.2 Motivation to Build Yet Another Language

Waves made a decision to build our own smart contract language for several reasons. Firstly, we want to build blockchain-friendly language, without loops or other possibilities for incorrect use of blockchain properties. Blockchain-friendliness is also a strong reason for not being Turing-complete, as we described before.

We want our language to be lazy. Lazy computations are able to save resources, by using them only when it is really necessary.

Ride is not supposed to be another JavaScript-like language, but we plan to expand the audience of users by using some modern developer-friendly syntax. The syntax of our language is functional, similar to Scala: strong and statically typed.

As a result, Ride offers a flexible but safe solution for on-blockchain computation.

4 Implementation

4.1 Ride as a non-Turing-complete language

Ride is a functional programming language based on expressions that are used for writing scripts. Ride's syntax is minimalistic. At the same time, the standard library includes a large number of useful functions.

Ride has strong static typing. It has no cycles, recursions, or goto-like expressions, and therefore is not Turing complete. Also, the language is conceived as lazy, which means that each variable's definition occurs only when this variable is used, but not before this. Lazy constant declaration is 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.fieldOne`). Calls to predefined functions are implemented via `()`.

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

With Ride, users can write functions or use it as a scripting language with special annotations, which consists of standard library version, content type, and script type. Functions can be annotated and used as domain-specific external interfaces.

4.2 Ride and Waves Blockchain

4.2.1 Waves Blockchain as an Environment

Waves has been built on top of an account-based model like Ethereum or Nxt (instead of Bitcoin's input/output system). There are two types of accounts: native accounts and smart accounts.

A native account has only data storage and some balance, and it can only make send transactions, signing them before sending them to the blockchain.

Since we have an account-based blockchain model, we can set security scripts on some accounts to make them smart. 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. To set a script on an account, a user has to send `SetScriptTransaction` with the corresponding compiled Ride script. Every account is allowed to become smart, or to change the script to null and become unscripted. For a more concrete example see section 7.1, with multi-signature code. All accounts have single data storage – storage on the account with the possibility to store an unlimited dictionary of useful data. To upload a data pair (key; value) a user can send a `DataTransaction` from her account with corresponding data.

In our platform we pay particular attention to creating tokens, so we have three types of token: a native one with fixed quantity (WAVES), issued native assets without a script, and scripted ones – smart assets. The token cannot become smart (or scripted) if this wasn't scripted initially with `IssueTransaction`. The smart asset script can be changed by `SetAssetScriptTransaction`, but only if this asset was scripted already. The example of whitelisting can be seen in section 7.2.

The main requirement for our smart accounts and smart assets is that they can be run for the price of normal transactions with predefined fees, without any additional "gas" or other costs. This is possible due to the statically predictable execution time. To change the state of the account (a combination of the account's data storage, the account's script and the account's balance) we also introduce a dApp element, with callable functions that are able to receive payments and send WAVES and tokens from the account as a script execution result. To initiate the call, the user has to use a new type of transaction: `InvokeScriptTransaction`. This needs to be put on the blockchain to call a special function from the dApp's account, and the sender pays fees to the miner for the invocation to be executed. The sender can optionally attach payment in WAVES or tokens, and upon invocation the contract state can be changed and the contract can make multiple payments. The dApp's account script can have several types of annotation. Annotations are supposed to be used for different domain purposes, for example to simplify the separation of functions: user functions, callable functions and a verifier function. The mechanics of smart accounts for authorization scripts will be maintained under the verifier function of the account with annotation `@Verifier`. The verifier function is a full analog of the standard script for a smart account: all outgoing transactions from this dApp account are checked by the function before they are put on the blockchain. This can be thought of as an "admin" function for the contract's owner or owners, and this is the reason why the dApp's script can have only one verifier function. By default, the contract, contract data and contract tokens are all controlled by the private key for the account. Multisig control is possible. If the verifier is always `false`, then the contract is sealed: immutable. The verifier function has no arguments, and the transaction is passed through annotation binding. The user's functions have no annotation and play the role of a normal function in a language that can be called from other functions to avoid duplication of code and simplify readability. The callable functions have an `@Callable` annotation and they are able to receive payments, to change account state and send tokens from the dApp's account. The dApp's account can have more than one callable function.The callable functions allow for atomic operation of multiple transfers and state changes. This is an essential feature for building token-based dApps such as stablecoins, etc. All the blockchain's users can call these functions. To call such a function, a user has to send `InvokeScriptTransaction` with certain parameters. In the callable function a user can check specific parameters; these values are passed through annotation binding of `@Callable(arg)` where it contains all the important information about invocation, for example, `payment` – the parameter of receiving tokens, the `caller` – the function's caller address, etc. If the function execution was successful, without exceptions in the execution process, then as an output the function returns a result in `ScriptResult` type. The `ScriptResult` type can be a set of data storage changes (`WriteSet`) and balance changes (`TransferSet`). As a script result a user can get only a `TransferSet` or a `WriteSet`, but they can be empty, in which case the function has no sense. The callable function execution can be interrupted by the `throw()` function in the case of exception; in this case transaction would be rejected and will not get to the blockchain. You can see an example of callable functions in section 7.3.

4.2.2 Lifecycle of a Contract on the Blockchain

There are several next stages in the lifecycle of our smart contract on the blockchain, which are presented in the Figure 1.

The first three stages, parsing, compilation and serialization, are off-chain. In the parsing stage, Ride text is parsed to an abstract syntax tree of expressions, then in the compilation stage, an abstract syntax tree is compiled to lower-level primitives. The compiler carries out type checking, verifies that all variables are in the right place, that the function calls get arguments of the correct type, etc. 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 subtype `BLOCK`. Each `EXPR` has a type and is one of:

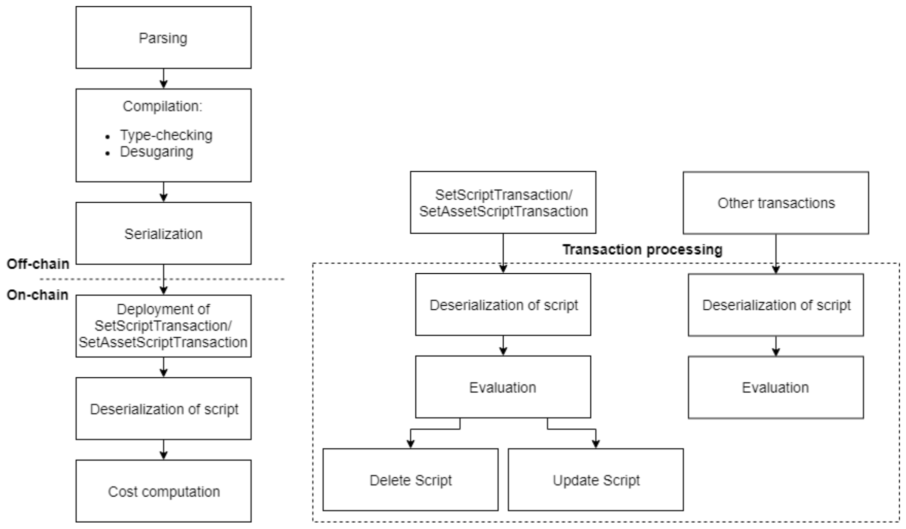
Parsing
Compilation:
Type-checking
Desugaring
Serialization
Off-chain
On-chain
Deployment of SetScriptTransaction/ SetAssetScriptTransaction
Deserialization of script
Cost computation
SetScriptTransaction/ SetAssetScriptTransaction
Other transactions
Transaction processing
Deserialization of script
Evaluation
Delete Script
Update Script
Deserialization of script
Evaluation

Figure 1: Smart contract lifecycle.

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

The compiled expression tree is serialized and sent to the network as part of `SetScriptTransaction`.

The following on-chain operations then occur: After deployment, during deserialization, we restore a compiled expression tree from bytes. Since Ride script is expression-based and has no loops or cycles, its computational complexity can be estimated before execution. If script complexity is greater than a defined cap (4,000 points), the transaction is rejected.

To calculate the estimate, each function and expression have costs defined in points. For example, the complexity of a sum operation (`+`) is 1, `sigVerify`'s complexity is 100 points, the complexity of `if-then-else` is the sum of the conditions' complexity and maximum between `then-` and `else-` expressions.

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.

Ride has no cycle and recursion possibility, unlike Solidity. We should note that Ride 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`s. 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 [8] it is shown that the 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.

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

```
match tx {
  case t:TransferTransaction => t.recipient
  case t:MassTransferTransaction => t.transfers
  case _ => throw()
}
```

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

4.3 Halting Problem

4.3.1 Termination of Parsing Stage

Let us prove that the parser always halts. As an input we have a byte array, which 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 of 10 types). Then a structure can be different and depends on the type.

4.3.2 Termination of Deserialization Stage

The deserialization stage builds an abstract syntax tree (AST), a directed acyclic graph, from script text.

A complete description of such a 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.

The formal representation of expression in Ride bytecode language is as follows:

$$
\begin{array}{l}
S \rightarrow T \\
T \rightarrow long \\
T \rightarrow string \\
T \rightarrow bytes \\
T \rightarrow true \\
T \rightarrow false \\
T \rightarrow if\ S\ S\ S \\
T \rightarrow block \\
block \rightarrow Definition\ S \\
Definition \rightarrow LET \\
Definition \rightarrow FUNC \\
LET \rightarrow String\ S \\
FUNC \rightarrow String,\ [String],\ S \\
T \rightarrow ref \\
ref \rightarrow string \\
T \rightarrow getter \\
getter \rightarrow S\ string \\
T \rightarrow funcall \\
funcall \rightarrow string \mid long\ ([S]) \\
DAPP \rightarrow\ [Definition],\ [CallableFunction], \\
\qquad\qquad VerifierFunction* \\
CallableFunction \rightarrow String,\ [String],\ S \\
VerifierFunction \rightarrow String,\ [String],\ S
\end{array}
$$

Note that `long`, `string`, `bytes`, `true` and `false` are terminals.

4.3.3 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 the 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:

```
let x = y
let y = x
x + y
```

In this fictional example, which will raise an error at the evaluation stage, we define `x` as `y` and viyce 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 them 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 `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.

4.3.4 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 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 the 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 the 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.

■

5 Add simulating loops with macro

Ride doesn't have any cycle/for/goto primitives because a decentralized network should not be able to evaluate an infinite algorithm. We therefore believe that the language should be non-Turing complete. But that fact can be frustrating for developers who need to traverse a collection, calculate the sum of an array, etc. This is why Ethereum has gas, or the ability to charge fees on a per-instruction basis at runtime. Our non-Turing completeness allows us to calculate computation costs upfront and reject not just the execution, but the deployment of a dApp itself, if necessary. Thus, as a solution for developers, we plan to add macros. Macros are just syntactic sugar, which curtail the amount of code that a developer has to write. When a compiler sees a macro, it rewrites the code it encloses and then compiles the output. One function existing in many languages is `fold`. Scala's version looks like this:

```
def fold[A](col: List[A], z: A, op: (A, A) => A): A$
```

where `z` is the value we start folding with, and then subsequently modify with `op` function and the next element in the collection. The `fold` function is extremely powerful: we can define, for example, sum, filter, zip, and exists without foreach . Let us give an example of sum realization:

```
func sum(a:Int, b:Int) = a + b
let arr = [1,2,3,4,5]
let sum = FOLD<5>(arr, 0, sum) # result: 15
```

This can be read as: make up to 5 steps; on each step take an accumulation result, the next element; apply `sum` function to the pair; return accumulation result. When compiling this, the compiler could unwrap:

```
let result = FOLD<5>(arr, acc0, function)
```

to:

```
let result = {
  let size = arr.size()
  if(size == 0) then acc0 else {
    let acc1 = function(acc0, arr[0])
    if(size == 1) then acc1 else {
      let acc2 = function(acc1, arr[1])
      if(size == 2) then acc2 else {
        let acc3 = function(acc2, arr[2])
        if(size == 3) then acc3 else {
          let acc4 = function(acc3, arr[3])
          if(size == 4) then acc4 else {
            let acc5 = function(acc4, arr[4])
            if(size == 5) then acc5 else
              throw("Too big array, max 5 elements")
}}}}}}
```

6 Formal verification

For any contract, formal verification will be particularly in demand, because it is impossible to correct a contract error after its launch. The price of such errors can be very high since smart accounts often store large amounts of funds.

Rides smart contracts are ideally suited to formal verification requirements. They are not Turing-complete, and their maximum complexity is limited by the lack of loops and `GOTO` statements. As soon as the Ride is a functional language, a systems state can be divided into two states: before scripts invocation and after it. This model, as the blockchain model as a whole, obeys temporal logic.

Any existing modelling and verification systems that may deal with temporal logic can be used for formal verification of smart contracts. In our appendix A.1 we use TLA+ for implementing the wallet example from section 7.3.

TLA+ is a is a formal specification language based on the set theory, first-order logic and temporal logic of action [9]. The language was developed by Leslie Lamport, a researcher of distributed systems theory. It is used to design, model, document, and verify concurrent systems.

The example in appendix A.1 is a full example of a wallet, but without certain details. For example, we do not check addition overflow and omit the blockchain model. We describe only the formal properties of the wallet model. It is an example of manual code translation from Ride to formal verification language, but in the future we do not exclude the possibility of creating an automatic translator for formal verification.

7 Examples section

7.1 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:

```
{-# STDLIB_VERSION 2 #-}
{-# CONTENT_TYPE EXPRESSION #-}
{-# SCRIPT_TYPE ACCOUNT #-}

#define public keys
let alicePubKey  = base58'5AzfA9UfpWVYiwFwvdr77k6LWupSTGLb14dEpMM'
let bobPubKey    = base58'2KwU4vzdgPmKyf7q354H9kSyX9NZjbnH2wi2VDF'
let cooperPubKey = base58'GbrUeGaBfmyFJjSQb9Z8uTCeXfRDVGJGrmgt5cD'

#check whoever provided the valid proof
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

#sum up every valid proof to get at least 2
aliceSigned + bobSigned + cooperSigned >= 2
```

Here users gather 3 signature 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.

7.2 White List Smart-Asset

To allow transfer of tokens only to specific accounts – to create a whitelist – you can use a smart asset with the following script, which checks for the presence of an address on the list:

```
  {-# CONTENT_TYPE EXPRESSION #-}
  {-# SCRIPT_TYPE ASSET #-}
  match tx {
    case t : TransferTransaction =>
      let trustedRecipient1 =
          addressFromString("3P6ms9EotRX8JwSrebeTXYVnzpCrKWLv4")
      let trustedRecipient2 =
          addressFromString("3PLZcCJyYQnfWfzhKXRA4rteC9J1ewf5K")
      let trustedRecipient3 =
          addressFromString("3PHrS6VNPRtUD8MHkfkmavL8JnGtSq5sx")

      t.recipient == trustedRecipient1
      || t.recipient == trustedRecipient2
      || t.recipient == trustedRecipient3

    case _ => false
  }
```

For purposes of security and provable completion of language, the list doesn't complete implementation of the iterator. Therefore, it is defined as a set of specific elements.

7.3 dApp: Wallet

In this example, a dApp wallet is implemented: you can send a WAVES payment, and it will save it in the wallet (deposit function), and you can take deposited WAVES back out of the wallet (withdraw function). All other transactions with this dApp account are prohibited. This is the reason why in the verifier function we always return `false`.

```
{-# STDLIB_VERSION 3 #-}
{-# CONTENT_TYPE DAPP #-}
{-# SCRIPT_TYPE ACCOUNT #-}

@Callable(i)
func deposit() = {
  let pmt = extract(i.payment)
  if (isDefined(pmt.assetId)) then throw("can hold waves only")
  else {
    let currentKey = toBase58String(i.caller.bytes)
    let currentAmount = match getInteger(this, currentKey) {
      case a:Int => a
      case _ => 0
    }

    let newAmount = currentAmount + pmt.amount
    WriteSet([DataEntry(currentKey, newAmount)])
  }
}

@Callable(i)
func withdraw(amount: Int) = {
  let currentKey = toBase58String(i.caller.bytes)
  let currentAmount = match getInteger(this, currentKey) {
    case a:Int => a
    case _ => 0
  }
  let newAmount = currentAmount - amount

  if (amount < 0)
    then throw("Can’t withdraw negative amount")
  else if (newAmount < 0)
    then throw("Not enough balance")
  else ScriptResult(
    WriteSet([DataEntry(currentKey, newAmount)]),
    TransferSet([ScriptTransfer(i.caller, amount, unit)])
  )
}

@Verifier(tx)
func verify() = false
```

References

[1] Szabo N. *Smart contracts: building blocks for digital markets.* EXTROPY: The Journal of Transhumanist Thought, (16), 1996.

[2] Szabo N. *The idea of smart contracts.*
Nick Szabos Papers and Concise Tutorials, 1997.

[3] Nakamoto S. et al. *Bitcoin: A peer-to-peer electronic cash system*, 2008.

[4] Buterin V. et al. *Ethereum's white paper*,
`https://github.com/ethereum/wiki/wiki/White-Paper`, 2013.

[5] Andrychowicz M. et al. *Secure Multiparty Computations on Bitcoin.* IEEE Symposium on Security and Privacy, 2014.

[6] Pass R. et al. *Micropayments for decentralized currencies.* Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, 207–218, 2015.

[7] Atzei N., Bartoletti M., Cimoli T. *A survey of attacks on Ethereum smart contracts.* International Conference on Principles of Security and Trust, 164–186, Springer, 2017.

[8] Chepurnoy A., Kharin V., Meshkov D. *Self-reproducing coins as universal Turing machine*, Data Privacy Management, Cryptocurrencies, and Blockchain Technology. Springer, Cham, 2018.

[9] Lamport, Leslie. *Specifying Concurrent Systems with TLA$^+$*, NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES, 173, 183–250, 1999.

A Appendices

A.1 Appendix TLA$^+$ Wallet

```
──────────────────────── MODULE wallet ────────────────────────

EXTENDS Integers, TLC
VARIABLES Balances, Data

InitBalances ≜ "Alice" :> 40 @@ "Bob" :> 20  @@ "wallet" :> 0
InitData ≜ [key ∈ {} ↦ {}]

Store(key, data) ≜ Data′ = [Data EXCEPT ![key] = data]
CheckData(key) ≜ key ∈ DOMAIN Data

Deposit(acc, amount) ≜ LET d ≜ IF CheckData(acc)
                                 THEN Data[acc]
                                 ELSE 0
                       IN   amount ≤ Balances[acc]
                        ∧   amount > 0
                        ∧   Data′ = acc :> d + amount @@ Data
                        ∧   IF acc = "wallet"
                             THEN
                              Balances′ = Balances
                             ELSE
                              Balances′ = [Balances EXCEPT ![acc] =
                                     Balances[acc] − amount, !["wallet"] =
                                     Balances["wallet"] + amount]

Withdraw(acc, amount) ≜ LET d ≜ IF CheckData(acc)
                                  THEN Data[acc]
                                  ELSE 0
                        IN   amount ≤ Balances["wallet"]
                         ∧   amount > 0
                         ∧   amount ≤ d
                         ∧   Data′ = acc :> d − amount @@ Data
                         ∧   IF acc = "wallet"
                              THEN
                               Balances′ = Balances
                              ELSE
                               Balances′ = [Balances EXCEPT ![acc] =
                                      Balances[acc] + amount, !["wallet"] =
                                      Balances["wallet"] − amount]


INSTANCE blockchain WITH
    InitBalances ← InitBalances,
    InitData ← InitData,
    Data ← Data,
    Balances ← Balances
```

$$\begin{aligned}
&\mathit{Next} \triangleq \exists\, \mathit{amount} \in 0\,..\,100,\ \mathit{acc} \in (\textsc{domain}\ \mathit{InitBalances}) \setminus \{\text{“wallet”}\} :\\
&\qquad \mathit{Deposit}(\mathit{acc},\ \mathit{amount}) \lor \mathit{Withdraw}(\mathit{acc},\ \mathit{amount})
\end{aligned}$$

$$\begin{aligned}
\mathit{WalletInvariant} \triangleq\ & \mathit{BlockchainInvariant}\\
\land\ & \mathit{TotalBalances}(\mathit{Data},\ (\textsc{domain}\ \mathit{Data}) \setminus \{\text{“wallet”}\}) = \mathit{Balances}[\text{“wallet”}]\\
\land\ & \forall\, \mathit{acc} \in (\textsc{domain}\ \mathit{InitBalances}) \setminus \{\text{“wallet”}\} : \textsc{if}\ \mathit{CheckData}(\mathit{acc})\\
& \qquad \textsc{then}\ \mathit{Data}[\mathit{acc}] \geq 0 \land \mathit{Balances}[\mathit{acc}] + \mathit{Data}[\mathit{acc}] = \mathit{InitBalances}[\mathit{acc}]\\
& \qquad \textsc{else}\ \ \textsc{true}\\
\land\ & \forall\, \mathit{acce} \in \{\mathit{acc1} \in \textsc{domain}\ \mathit{Data} : \mathit{Data}[\mathit{acc1}] > 0\}\\
& \quad : \textsc{enabled}\ (\mathit{Withdraw}(\mathit{acce},\ \mathit{Data}[\mathit{acce}]))
\end{aligned}$$