scilla-doc Documentation

Release 0.0.1

Amrit Kumar

Dec 08, 2018

Contents

Scilla short for *Smart Contract Intermediate-Level LAnguage* is an intermediate-level smart contract language being developed for Zilliqa. Scilla has been designed as a principled language with smart contract safety in mind.

Scilla imposes a structure on smart contracts that will make applications less vulnerable to attacks by eliminating certain known vulnerabilities directly at the language-level. Furthermore, the principled structure of Scilla will make applications inherently more secure and amenable to formal verification.

The language is being developed hand-in-hand with formalization of its semantics and its embedding into the Coq proof assistant — a state-of-the art tool for mechanized proofs about properties of programs. Coq is based on advanced dependently-typed theory and features a large set of mathematical libraries. It has been successfully applied previously to implement certified (i.e., fully mechanically verified) compilers, concurrent and distributed applications, including blockchains among others.

Zilliqa — the underlying blockchain platform on which Scilla contracts are run has been designed to be scalable. It employs the idea of sharding to validate transactions in parallel. Zilliqa has an intrinsic token named *Zilling*, ZIL for short that are required to run smart contracts on Zilliqa.

CHAPTER 1

Development Status

Scilla is under active research and development and hence parts of the specification described in this document are subject to change. Scilla currently comes with an interpreter binary that has been integrated into two Scilla-specific web-based IDEs. *Trying out Scilla* presents the features of the two IDEs.

Note that Scilla does not have a type checker implemented yet and hence type safety of contracts written in Scilla is not guaranteed.

CHAPTER 2

Resources

There are several resources to learn about Scilla and Zilliqa. Some of these are given below:

Scilla

- Scilla Design Document
- Scilla Slides
- Scilla Language Grammar
- Scilla Design Story Piece by Piece: Part 1 (Why do we need a new language?)

Zilliqa

- The Zilliqa Design Story Piece by Piece: Part 1 (Network Sharding)
- The Zilliqa Design Story Piece by Piece: Part 2 (Consensus Protocol)
- The Zilliqa Design Story Piece by Piece: Part 3 (Making Consensus Efficient)
- Technical Whitepaper
- The Not-So-Short Zilliqa Technical FAQ

CHAPTER 3

Contents

3.1 Scilla Design Principles

Smart contracts provide a mechanism to express computations on a blockchain, i.e., a decentralized Byzantine-fault tolerant distributed ledger. With the advent of smart contracts, it has become possible to build what is referred to as *decentralized applications* or Dapps for short. These applications have their program and business logic coded in the form of a smart contract that can be run on a decentralized blockchain network.

Running applications on a decentralized network eliminates the need of a trusted centralized party or a server typical of other applications. These features of smart contracts have become so popular today that they now drive real-world economies through applications such as crowdfunding, games, decentralized exchanges, payment processors among many others.

However, experience over the last few years has shown that implemented operational semantics of smart contract languages admit rather subtle behaviour that diverge from the *intuitive understanding* of the language in the minds of contract developers. This divergence has led to some of the largest attacks on smart contracts, e.g., the attack on the DAO contract and Parity wallet among others. The problem becomes even more severe because smart contracts cannot directly be updated due to the immutable nature of blockchains. It is hence crucial to ensure that smart contracts that get deployed are safe to run.

Formal methods such as verification and model checking have proven to be effective in improving the safety of software systems in other disciplines and hence it is natural to explore their applicability in improving the readability and safety of smart contracts. Moreover, with formal methods, it becomes possible to produce rigorous guarantees about the behavior of a contract.

Applying formal verification tools with existing languages such as Solidity however is not an easy task because of the extreme expressivity typical of a Turing-complete language. Indeed, there is a trade-off between making a language simpler to understand and amenable to formal verification and making it more expressive. For instance, Bitcoin's scripting language occupies the *simpler* end of the spectrum and does not handle stateful-objects. On the *expressive* side of the spectrum is a Turing-complete language such as Solidity.

Scilla is a new (intermediate-level) smart contract language that has been designed to achieve both *expressivity* and *tractability* at the same time, while enabling formal reasoning about contract behavior by adopting certain fundamental design principles as described below:

Separation Between Computation and Communication

Contracts in Scilla are structured as communicating automata: every in-contract computation (e.g., changing its balance or computing a value of a function) is implemented as a standalone, atomic transition, i.e., without involving any other parties. Whenever such involvement is required (e.g., for transferring control to another party), a transition would end, with an explicit communication, by means of sending and receiving messages. The automata-based structure makes it possible to disentangle the contract-specific effects (i.e., transitions) from blockchain-wide interactions (i.e., sending/receiving funds and messages), thus providing a clean reasoning mechanism about contract composition and invariants.

Separation Between Effectful and Pure Computations

Any in-contract computation happening within a transition has to terminate, and have a predictable effect on the state of the contract and the execution. In order to achieve this, Scilla draws inspiration from functional programming with effects, drawing a distinction between pure expressions (e.g., expressions with primitive data types and maps), impure local state manipulations (i.e., reading/writing into contract fields) and blockchain reflection (e.g., reading current block number). By carefully designing semantics of interaction between pure and impure language aspects, Scilla ensures a number of foundational properties about contract transitions, such as progress and type preservation, while also making them amenable to interactive and/or automatic verification with standalone tools.

Separation Between Invocation and Continuation

Structuring contracts as communicating automata provides a computational model, which only allows *tail-calls*, i.e., every call to an external function (i.e., another contract) has to be done as the absolutely last instruction.

3.2 Trying out Scilla

Scilla is under active development. At this stage, there are two ways to try out Scilla.

3.2.1 Savant IDE

Savant IDE is a web-based development environment that is not connected to any external blockchain network. It hence simulates a blockchain in the browser's memory by maintaining persistent account states. It is optimized for use in Chrome Web Browser.

Users will not need to hold testnet ZIL to use Savant, instead they are given 20 arbitrary accounts with 1,000,000,000 fake ZILs to test their contracts.

Savant serves as a staging environment, before doing automated script testing with tools like Kaya (TestRPC) and Javascript library. To try out the Savant IDE, users need to visit Savant IDE.

3.2.2 Blockchain IDE

The other way try out Scilla is through the Scilla Blockchain IDE. The IDE is connected to the Zilliqa blockchain via a testnet wallet and a block explorer and hence comes with (almost) all the features needed to test a Scilla contract in a real blockchain environment.

In order to use the Scilla Blockchain IDE, users will have to hold Testnet ZIL (tokens to use Zilliqa's blockchain infrastructure). Testnet ZIL tokens are required to pay for gas fees to deploy and run smart contracts. These tokens are periodically distributed for free.

To try out the Blockchain IDE, users need to go through the Zilliqa testnet wallet.

3.2.3 Example Contracts

Both IDEs come with the following sample smart contracts written in Scilla:

- **HelloWorld** : It is a simple contract that allows a specified account denoted `owner` to set a welcome message. Setting the welcome message is done via `setHello (msg: String)`. The contract also provides an interface `getHello()` to allow any account to be returned with the welcome message when called.
- **CrowdFunding** : Crowdfunding implements a kickstarter campaign where users can donate funds to the contract using `Donate()`. If the campaign is successful, i.e., enough money is raised within a given time period, the raised money can be sent to a pre-defined account `owner` via `GetFunds()`. Else, if the campaign fails, then contributors can take back their donations via the transition `ClaimBack()`.
- **ZilGame** : It is a two-player game where the goal is to find the closest pre-image of a given SHA256 digest (`puzzle`). More formally, given a digest *d*, and two values *x* and *y*, *x* is said to be a closer pre-image than *y* of *d* if Distance(SHA-256(x), d) < Distance(SHA-256(y), d), for some *Distance* function. The game is played in two phases. In the first phase, players submit their hash, i.e., SHA-256(x) and SHA-256(y) using the transition `Play(guess: ByStr32)`. Once the first player has submitted her hash, the second player has a bounded time to submit her hash. If the second player does not submit her hash within the stipulated time, then the first player may become the winner. In the second phase, players have to submit the corresponding values `x` or `y` using the transition `ClaimReward(solution: Int128)`. The player submitting the closest pre-image is declared the winner and wins a reward. The contract also provides a transition `Withdraw ()` to recover funds and send to a specified `owner` in case no player plays the game.
- **FungibleToken** : Fungible token contract that mimics an ERC20 style fungible token standard. Defacto standard for tokenised utility tokens.
- **NonFungible Token** : Non fungible token contract that mimics an ERC721 style NFT token standard for unique tokenised assets. Example use case could be in-game items like CryptoKitties.
- **OpenAuction** : A simple open auction contract where bidders can make their bid using `Bid()`, and the highest and winning bid amount goes to a pre-defined account. Bidders who don't win can take back their bid using the transition `Withdraw()`. The organizer of the auction can claim the highest bid by invoking the transition `AuctionEnd()`.
- **BookStore** : A demonstration of a CRUD app. Only `owner` of the contract can add `members`. All `members` will have read/write access capability to create OR update books in the inventory with *book title*, *author*, and *bookID*.
- **SchnorrTest** : A sample contract to test the generation of a Schnorr public/private keypairs, signing of a `msg` with the private keys, and verification of the signature.

3.3 Scilla by Example

3.3.1 HelloWorld

We start off by writing a classical `HelloWorld.scilla` contract with the following specification:

- It should have an *immutable variable* `owner` to be initialized by the creator of the contract. The variable is immutable in the sense that once initialized, its value cannot be changed. `owner` will be of type `ByStr20` (a hexadecimal Byte String representing 20 bytes).
- It should have a *mutable variable* `welcome_msg` of type `String` initialized to `""`. Mutability here refers to the possibility of modifying the value of a variable even after the contract has been deployed.

- The `owner` and **only her** should be able to modify `welcome_msg` through an interface `setHello`. The interface takes a `msg` (of type `String`) as input and allows the `owner` to set the value of `welcome_msg` to `msg`.
- It should have an interface `getHello` that welcomes any caller with `welcome_msg`. `getHello` will not take any input.

Defining Contract and its (Im)Mutable Variables

A contract is declared using the `contract` keyword that starts the scope of the contract. The keyword is followed by the name of the contract which will be `HelloWorld` in our example. So, the following code fragment declares a `HelloWorld` contract.

```
contract HelloWorld
```

Note: In the current implementation, a Scilla contract can only contain a single contract declaration and hence any code that follows the `contract` keyword is part of the contract declaration. In other words, there is no explicit keyword to declare the end of the contract definition.

A contract declaration is followed by the declaration of its immutable variables, the scope of which is defined by `()`. Each immutable variable is declared in the following way: `vname: vtype`, where `vname` is the variable name and `vtype` is the variable type. Immutable variables are separated by `,`. As per the specification, the contract will have only one immutable variable `owner` of type `ByStr20` and hence the following code fragment.

```
(owner: ByStr20)
```

Mutable variables in a contract are declared through keyword `field`. Each mutable variable is declared in the following way: `field vname : vtype = init_val`, where `vname` is the variable name, `vtype` is its type and `init_val` is the value to which the variable has to be initialized. The `HelloWorld` contract has one mutable parameter `welcome_msg` of type `String` initialized to `""`. This yields the following code fragment:

```
field welcome_msg : String = ""
```

At this stage, our `HelloWorld.scilla` contract will have the following form that includes the contract name and its (im)mutable variables:

```
contract HelloWorld
(owner: ByStr20)

field welcome_msg : String = ""
```

Note: In addition to these fields, any contract in Scilla has an implicitly declared mutable field `_balance` (initialised upon the contract's creation), which keeps the amount of funds held by the contract. This field can be freely read within the implementation, but can only modified by explicitly transferring funds to other accounts.

Defining Interfaces *aka* Transitions

Interfaces like `setHello` are referred to as *transitions* in Scilla. Transitions are similar to *functions* or *methods* in other languages.

Note: The term *transition* comes from the underlying computation model in Scilla which follows a communicating automaton. A contract in Scilla is an automaton with some state. The state of an automaton can be changed via a transition that takes a previous state and an input and yields a new state. Check wikipedia entry to read more about transition systems.

A transition is declared using the keyword `transition`. The end of a transition scope is declared using the keyword `end`. The `transition` keyword is followed by the transition name, which is `setHello` for our example. Then follows the input parameters within `()`. Each input parameter is separated by a `,` and is declared in the following format: `vname : vtype`. According to the specification, `setHello` takes only one parameter of name `msg` of type `String`. This yields the following code fragment:

```
transition setHello (msg : String)
```

What follows the transition signature is the body of the transition. Code for the first transition `setHello (msg : String)` to set `welcome_msg` is given below:

```
1  transition setHello (msg : String)
2    is_owner = builtin eq owner _sender;
3    match is_owner with
4    | False =>
5      msg = {_tag : "Main"; _recipient : _sender; _amount : Uint128 0; code : not_owner_
   ↪code};
6      msgs = one_msg msg;
7      send msgs
8    | True =>
9      welcome_msg := msg;
10     msg = {_tag : "Main"; _recipient : _sender; _amount : Uint128 0; code : set_hello_
   ↪code};
11     msgs = one_msg msg;
12     send msgs
13   end
14 end
```

At first, the caller of the transition is checked against the `owner` using the instruction `builtin eq owner _sender` in `Line 2`. In order to compare two addresses, we are using the function `eq` defined as a `builtin` operator. The operator returns a boolean value `True` or `False`.

Note: Scilla internally defines some variables that have special semantics. These special variables are often prefixed by _. For instance, `_sender` in Scilla refers to the account address that called the current contract.

Depending on the output of the comparison, the transition takes a different path declared via *pattern matching*, the syntax of which is given in the fragment below.

```
match expr with
| x => expr_1
| y => expr_2
end
```

The above code checks whether `expr` evaluates to `x` or `y`. If `expr` evaluates to `x`, then the next expression to be evaluated will be `expr_1`, else if it evaluates to `y`, then, the next expression to be evaluated will be `expr_2`. Simply put, the above code implements an `if-then-else` instruction.

Caller is not owner

In case the caller is different from `owner`, the transition takes the `False` branch and the contract sends out a message. Scilla defines a special type `Message` for outgoing messages. An outgoing message contains information about any other contract that needs to be called as a part of the current call.

The output message in this case is an error code `not_owner_code` included in `msg`. More concretely, the output message in this case is:

```
msg = {_tag : "Main"; _recipient : _sender; _amount : Uint128 0; code : not_owner_
↪code};
```

An outgoing message is formed of `vname : value` pairs delimited by `;`, the scope of which is defined by `{}`. Each outgoing message must have three compulsory fields: `_tag`, `_recipient` and `_amount` in no particular order. `_recipient` is an account address to which the message will be sent. `_tag` is the name of the transition to be invoked in `_recipient` and `_amount` is the number of ZIL to be transferred to `_recipient`.

Apart from these compulsory fields, a message may have other fields. In the current example, the message has a field `code` to report an error message.

Sending a message out is done using the `send` instruction that takes a list of entries of type `Message`. In the current example, the list will contain only one entry. To sum up, the following code will create a message and send it out.

```
msgs = one_msg msg;
send msgs
```

`one_msg` is a utility function that allows to create a list of messages and inserts `msg` into the list.

Caller is owner

In case the caller is `owner`, the contract allows the caller to set the value of the mutable variable `welcome_msg` to the input parameter `msg`. It is done through the following instruction.

```
welcome_msg := msg;
```

Note: Writing to a mutable parameter is done via the operator `:=`.

And as in the previous case, the contract then sends out a message to the caller with the code `set_hello_code`.

Libraries

A Scilla contract may come with some helper libraries that declare purely functional (with no state manipulation) components of a contract. A library is declared in the preamble of a contract using the keyword `library` followed by the name of the library. In our current example a library declaration would look like the following:

```
library HelloWorld
```

In our example, the library will include the definition of the error codes as given below defined using standard `let x = y in expr` construct.

```
let not_owner_code  = Uint32 1
let set_hello_code  = Uint32 2
```

The library may also include utility functions, for instance, the function `one_msg` that creates a list with one entry of type `Message` as given below:

```
let one_msg =
   fun (msg : Message) =>
   let nil_msg = Nil {Message} in
   Cons {Message} msg nil_msg
```

At this stage, our contract fragment will have the following form:

```
library HelloWorld

 let one_msg =
     fun (msg : Message) =>
     let nil_msg = Nil {Message} in
     Cons {Message} msg nil_msg

 let not_owner_code  = Uint32 1
 let set_hello_code  = Uint32 2


 contract HelloWorld
 (owner: ByStr20)

 field welcome_msg : String = ""

 transition setHello (msg : String)
   is_owner = builtin eq owner _sender;
   match is_owner with
   | False =>
     msg = {_tag : "Main"; _recipient : _sender; _amount : Uint128 0; code : not_
↪owner_code};
     msgs = one_msg msg;
     send msgs
   | True =>
     welcome_msg := msg;
     msg = {_tag : "Main"; _recipient : _sender; _amount : Uint128 0; code : set_
↪hello_code};
     msgs = one_msg msg;
     send msgs
   end
 end
```

Final Touches

We may now add the second transition `getHello()` that allows any caller to be greeted by `welcome_msg`. The declaration is similar to `setHello (msg : String)` accept that `getHello()` does not take any parameter.

```
transition getHello ()
    r <- welcome_msg;
    msg = {_tag : Main; _recipient : _sender; _amount : 0; msg : r};
    msgs = one_msg msg;
    send msgs
end
```

Note: Reading from a mutable variable is done via the operator `<-`. In our example, this translates to `r <-`

`welcome_msg`.

The complete contract that implements the desired specification is given below:

```
(* HelloWorld contract *)


(***************************************************)
(*               Associated library                *)
(***************************************************)
library HelloWorld

let one_msg =
  fun (msg : Message) =>
  let nil_msg = Nil {Message} in
  Cons {Message} msg nil_msg

let not_owner_code  = Uint32 1
let set_hello_code  = Uint32 2

(***************************************************)
(*             The contract definition             *)
(***************************************************)

contract HelloWorld
(owner: ByStr20)

field welcome_msg : String = ""

transition setHello (msg : String)
  is_owner = builtin eq owner _sender;
  match is_owner with
  | False =>
    msg = {_tag : "Main"; _recipient : _sender; _amount : 0; code : not_owner_code};
    msgs = one_msg msg;
    send msgs
  | True =>
    welcome_msg := msg;
    msg = {_tag : "Main"; _recipient : _sender; _amount : 0; code : set_hello_code};
    msgs = one_msg msg;
    send msgs
  end
end

transition getHello ()
    r <- welcome_msg;
    msg = {_tag : Main; _recipient : _sender; _amount : 0; msg : r};
    msgs = one_msg msg;
    send msgs
end
```

3.3.2 Crowdfunding

In this section, we present a slightly more involved contract that runs a crowdfunding campaign. In a crowdfunding campaign, a project owner wishes to raise funds through donations from the community.

It is assumed that the owner (`owner`) wishes to run the campaign for a certain pre-determined period of time

(max_block). The owner also wishes to raise a minimum amount of funds (goal) without which the project can not be started. The contract hence has three immutable variables owner, max_block and goal.

The campaign is deemed successful if the owner can raise the minimum goal in the stipulated time. In case the campaign is unsuccessful, the donations are returned to the project backers who contributed during the campaign. The contract maintains two mutable variables: backer a map between contributor's address and amount contributed and a boolean flag funded that indicates whether the owner has already transferred the funds after the end of the campaign.

The contract contains three transitions: Donate () that allows anyone to contribute to the crowdfunding campaign, GetFunds () that allows **only the owner** to claim the donated amount and transfer it to owner and ClaimBack() that allows contributors to claim back their donations in case the campaign is not successful.

The complete contract is given below:

```
(***************************************************)
(*               Associated library                *)
(***************************************************)
library Crowdfunding

let andb =
  fun (b : Bool) =>
  fun (c : Bool) =>
    match b with
    | False => False
    | True  =>
      match c with
      | False => False
      | True  => True
      end
    end

let orb =
  fun (b : Bool) => fun (c : Bool) =>
    match b with
    | True  => True
    | False =>
      match c with
      | False => False
      | True  => True
      end
    end

let negb = fun (b : Bool) =>
  match b with
  | True => False
  | False => True
  end

let one_msg =
  fun (msg : Message) =>
    let nil_msg = Nil {Message} in
    Cons {Message} msg nil_msg

let check_update =
  fun (bs : Map ByStr20 Uint128) =>
  fun (_sender : ByStr20) =>
  fun (_amount : Uint128) =>
    let c = builtin contains bs _sender in
    match c with
```

(continues on next page)

(continued from previous page)

```
    | False =>
      let bs1 = builtin put bs _sender _amount in
      Some {Map ByStr20 Uint128} bs1
    | True  => None {Map ByStr20 Uint128}
    end

let blk_leq =
  fun (blk1 : BNum) =>
  fun (blk2 : BNum) =>
    let bc1 = builtin blt blk1 blk2 in
    let bc2 = builtin eq blk1 blk2 in
    orb bc1 bc2

let accepted_code = Uint32 1
let missed_deadline_code = Uint32 2
let already_backed_code  = Uint32 3
let not_owner_code  = Uint32 4
let too_early_code  = Uint32 5
let got_funds_code  = Uint32 6
let cannot_get_funds  = Uint32 7
let cannot_reclaim_code = Uint32 8
let reclaimed_code = Uint32 9

(***************************************************)
(*             The contract definition             *)
(***************************************************)
contract Crowdfunding

(*  Parameters *)
(owner     : ByStr20,
 max_block : BNum,
 goal      : Uint128)

(* Mutable fields *)
field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
field funded : Bool = False

transition Donate ()
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs _sender _amount;
    match res with
    | None =>
      msg  = {_tag : Main; _recipient : _sender; _amount : 0;
              code : already_backed_code};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {_tag : Main; _recipient : _sender; _amount : 0;
              code : accepted_code};
      msgs = one_msg msg;
      send msgs
```

(continues on next page)

(continued from previous page)

```
    end
  | False =>
    msg  = {_tag : Main; _recipient : _sender; _amount : 0;
            code : missed_dealine_code};
    msgs = one_msg msg;
    send msgs
  end
end

transition GetFunds ()
  is_owner = builtin eq owner _sender;
  match is_owner with
  | False =>
    msg  = {_tag : Main; _recipient : _sender; _amount : 0;
            code : not_owner_code};
    msgs = one_msg msg;
    send msgs
  | True =>
    blk <- & BLOCKNUMBER;
    in_time = blk_leq blk max_block;
    c1 = negb in_time;
    bal <- _balance;
    c2 = builtin lt bal goal;
    c3 = negb c2;
    c4 = andb c1 c3;
    match c4 with
    | False =>
      msg  = {_tag : Main; _recipient : _sender; _amount : 0;
              code : cannot_get_funds};
      msgs = one_msg msg;
      send msgs
    | True =>
      tt = True;
      funded := tt;
      msg  = {_tag : Main; _recipient : owner; _amount : bal;
              code : got_funds_code};
      msgs = one_msg msg;
      send msgs
    end
  end
end

(* transition ClaimBack *)
transition ClaimBack ()
  blk <- & BLOCKNUMBER;
  after_deadline = builtin blt max_block blk;
  match after_deadline with
  | False =>
    msg  = {_tag : Main; _recipient : _sender; _amount : 0;
        code : too_early_code};
    msgs = one_msg msg;
    send msgs
  | True =>
    bs <- backers;
    bal <- _balance;
    (* Goal has not been reached *)
    f <- funded;
```

(continues on next page)

(continued from previous page)

```
    c1 = builtin lt bal goal;
    c2 = builtin contains bs _sender;
    c3 = negb f;
    c4 = andb c1 c2;
    c5 = andb c3 c4;
    match c5 with
    | False =>
      msg  = {_tag : Main; _recipient : _sender; _amount : 0;
              code : cannot_reclaim_code};
      msgs = one_msg msg;
      send msgs
    | True =>
      res = builtin get bs _sender;
      match res with
      | None =>
        msg  = {_tag : Main; _recipient : _sender; _amount : 0;
                code : cannot_reclaim_code};
        msgs = one_msg msg;
        send msgs
      | Some v =>
        bs1 = builtin remove bs _sender;
        backers := bs1;
        msg  = {_tag : Main; _recipient : _sender; _amount : v;
                code : reclaimed_code};
        msgs = one_msg msg;
        send msgs
      end
    end
  end
end
```

3.4 Scilla in Depth

3.4.1 Structure of a Scilla Contract

The general structure of a Scilla contract is given in the code fragment below:

- It starts with the declaration of a `library` that contains purely mathematical functions. For instance, a function to compute the boolean `AND` of two bits or computing factorial of a given natural number.
- Then, follows the actual contract definition declared using the keyword `contract`.
- Within a contract, there are then three distinct parts:
  1. The first part declares the immutable parameters of the contract.
  2. The second part declares the mutable fields.
  3. The third part contains all `transition` definitions.

```
(* Scilla contract structure *)


(***************************************************)
(*               Associated library                *)
(***************************************************)
```

(continues on next page)

(continued from previous page)

```
library MyContractLib


(* Library code block follows *)



(***************************************************)
(*             Contract definition                 *)
(***************************************************)

contract MyContract

(* Immutable fields declaration *)

(vname_1 : vtype_1,
 vname_2 : vtype_2)

(* Mutable fields declaration *)

field vname_1 : vtype_1 = init_val_1
field vname_2 : vtype_2 = init_val_2

(* Transitions *)


(* Transition signature *)
transition firstTransition (param_1 : type_1, param_2 : type_2)
  (* Transition body *)

end

transition secondTransition (param_1: type_1)
  (* Transition body *)

end
```

Immutable Variables

Immutable variables, are the contract's initial parameters, whose values are defined at the time of contract creation and cannot be modified after.

Declaration of immutable variables has the following format:

```
(vname_1 : vtype_1,
 vname_2 : vtype_2,
  ...  )
```

Each declaration consists of a variable name (an identifier) and followed by its type, separated by `:`. Multiple variable declarations are separated by `,`. The initialization values for variables are to be specified at the time of contract creation.

Mutable Variables

Mutable variables represent the mutable state of the contract. They are also called *fields*. They are declared after the immutable variables, with each declaration prefixed with the keyword `field`.

```
field vname_1 : vtype_1 = expr_1
field vname_2 : vtype_2 = expr_2
...
```

Each expression here is an initializer for that value. The definitions complete the initial state of the contract, at the time of creation. As the contract goes through transitions, the values of these fields get modified.

Transitions

Transitions define the change in the state of the contract. These are defined with the keyword `transition` followed by the parameters to be passed. The definition ends with the `end` keyword.

```
transition foo (vname_1 : vtype_1, vname_2 : vtype_2, ...)
  ...
end
```

where `vname : vtype` specifies the name and type of each parameter and multiple parameters are separated by `,`.

Note: In addition to parameters that are explicitly declared in the definition, each `transition` has available to it, the following implicit parameters:

- `_sender : ByStr20` : The account address that triggered this transition. In case, the transition was called by a contract account instead of a user account, then `_sender` is the contract address.
- `_amount : Uint128` : Incoming amount (ZILs) sent by the sender. This amount must be explicitly accepted using the `accept` statement within the transition. The money transfer does not happen if the transition does not execute `accept`.

Expressions

Expressions handle pure operations. The supported expressions in Scilla are:

- `let x = f` : Give `f` the name `x` in the contract. The binding of `x` to `f` is **global** and extends to the end of the contract. The following code fragment defines a constant `one` whose values is `1` of type `Int32` throughout the contract.

  ```
  let one = Int32 1
  ```

- `let x = f in expr` : Bind `f` to the name `x` within expression `expr`. The binding here is **local** to `expr` only. The following example binds the value of `one` to `1` of type `Int32` and `two` to `2` of type `Int32` in the expression `builtin add one two`, which adds `1` to `2` and hence evaluates to `3` of type `Int32`.

  ```
  let sum =
    let one = Int32 1 in
    let two = Int32 2 in
    builtin add one two
  ```

- `{ <entry>_1 ; <entry>_2 ... }`: Message expression, where each entry has the following form: `b : x`. Here `b` is an identifier and `x` a variable, whose value is bound to the identifier in the message.
- `fun (x : T) => expr` : A function that takes an input `x` of type `T` and returns the value to which expression `expr` evaluates.
- `tfun T => expr` : A type function that takes `T` as a parametric type and returns the value to which expression `expr` evaluates. These are typically used to build library functions. See the section on *Pairs* below for an example.
- `@x T`: Instantiate a variable `x` with type `T`.
- `f x` : Apply `f` on `x`.
- `builtin f x`: Apply the `builtin` function `f` on `x`.
- `match` expression: Matches a bound variable with patterns and executes the statements in that clause. The `match` expression is similar to the `match` in OCaml. The pattern to be matched can be a variable binding, an ADT constructor (see *ADTs*) or the wildcard _ symbol to match anything.

```
match x with
| pattern_1 =>
  statements ...
| pattern_2 =>
  statements ...
| _ => (*Wildcard*)
  statements ...
end
```

Statements

Statements in Scilla are operations with effect, i.e., these operations are impure and hence not purely mathematical. Such operations including reading or writing from/to a mutable smart contract variable.

- `x <- f` : Read from a mutable field `f` into `x`.
- `f := x` : Update mutable field `f` with value `x`.

One can also read from the blockchain state. A blockchain state consists of certain values associated with a block, for instance, the `BLOCKNUMBER`.

- `x <- & BLOCKNUMBER` reads from the blockchain state variable `BLOCKNUMBER` into `x`.

Whenever ZIL tokens are sent via a transition, the transition has to explicitly accept the transfer. This is done through the `accept` statement.

- `accept` : Accept incoming payment.

Communication

A contract can communicate with other contracts (or non-contract) accounts through `send` statement:

- `send msgs` : send a list of messages `msgs`.

  The following code defines a `msg` with four entries `_tag`, `_recipient`, `_amount` and `param`. `_tag` identifier entry is used to identify the name of the next transition to be executed in `_recipient`, while `_amount` is the number of ZILs to be transferred to `_recipient`, where, `param` is any parameter to be passed to the transition.

```
(*Assume contractAddress is the address of the contract being called and the␣
↪contract contains the transition setHello*)
msg = { _tag : "setHello"; _recipient : contractAddress; _amount : Uint128 0;␣
↪param : Uint32 0 };
```

Every message must have `_tag`, `_recipient` and `_amount` entries.

A contract can also communicate to the client (off-chain) by emitting events:

- `event e`: emit an event `e`. The following code emits an event with name `eventName`.

```
e = { _eventname : "eventName"; <entry>_2 ; <entry>_3 };
(*where <entry> is of the form: b : x as in a message expression.*)
(*Here b is the identifier, and x the variable, whose value is bound to the
identifier.*)
event e;
```

Note that the first entry is always `_eventname` and is compulsory.

3.4.2 Primitive Data Types & Operations

Integer Types

Scilla defines signed and unsigned integer types of 32, 64, 128, and 256 bits. These integer types can be specified with the keywords `IntX` and `UintX` where `X` can be 32, 64, 128, or 256. For example, an unsigned integer of 32 bits can be specified as `Uint32`.

The following code snippet declares a global `Uint32` integer:

```
let x = Uint32 43
```

The following operations on integers are language built-ins. Each operation takes two integers `IntX`/`UintX` (of the same type) as arguments.

- `builtin eq i1 i2` : Is `i1` equal to `i2` Returns `Bool`.
- `builtin add i1 i2`: Add integer values `i1` and `i2`. Returns an integer of the same type.
- `builtin sub i1 i2`: Subtract `i2` from `i1`. Returns an integer of the same type.
- `builtin mul i1 i2`: Integer product of `i1` and `i2`. Returns an integer of the same type.
- `builtin div i1 i2`: Integer division of `i1` by `i2`. Returns an integer of the same type.
- `builtin rem i1 i2`: `i1` modulo `i2`. Returns an integer of the same type.
- `builtin lt i1 i2`: Is `i1` lesser than `i2`. Returns `Bool`.

Note: Values related to money (such as amount transferred or the balance of an account) are `Uint128`.

Strings

As with most languages, `String` literals in Scilla are expressed using a sequence of characters enclosed in double quotes. Variables can be declared by specifying using keyword `String`.

The following code snippet declares a global `String` constant:

```
let x = "Hello"
```

The following `String` operations are language built-ins.

- `builtin eq s1 s2` : Is `String s1` equal to `String s2`. Returns `Bool`.
- `builtin concat s1 s2` : Concatenate `String s1` with `String s2`. Returns `String`.
- `builtin substr s1 i1 i2` : Extract sub-string of `String s1` starting from position `Uint32 i1` with length `Uint32 i2`. Returns `String`.

Hashes

A hash in Scilla is declared using the data type `ByStr32`. A `ByStr32` represents a hexadecimal Byte String of 32 bytes (64 hexadecimal characters) prefixed with `0x`.

The following code snippet declares a global `ByStr32` constant:

```
let x = 0x1234567890123456789012345678901234567890123456789012345678901234567890abff
```

The following operations on hashes are language built-ins. In the description below, `Any` can be of type `IntX`, `UintX`, `String`, `ByStr20` or `ByStr32`.

- `builtin eq h1 h2`: Is `ByStr32 h1` equal to `ByStr32 h2`. Returns `Bool`.
- `builtin dist h1 h2`: The distance between `ByStr32 h1` and `ByStr32 h2`. Returns `Uint256`.
- `builtin sha256hash x` : The SHA256 hash of value of x of type `Any`. Returns `ByStr32`.
- `builtin keccak256hash x`: The Keccak256 hash of a value of x of type `Any`. Returns `ByStr32`.
- `builtin ripemd160hash x`: The RIPEMD-160 hash of a value of x of type `Any`. Returns `ByStr16`.
- `builtin to_byStr x'` : Converts a hash `x'` of finite length, say of type `ByStr32` to one of arbitrary length.
- `builtin schnorr_gen_key_pair` : Create a key pair of form `Pair {ByStr32 BySt33}` that consist of both private key of type `ByStr32` and public key of type `ByStr33` respectively.
- `builtin schnorr_sign privk msg` : Sign a `msg` of type `ByStr` with the `privk` of type `ByStr32`.
- `builtin schnorr_verify pubk msg sig` : Verify a signed `sig` of type `ByStr64` against the `msg` of type `ByStr32` with the `pubk` of type `ByStr33`.

Maps

`Map` values provide key-value store. Keys can have types `IntX`, `UintX`, `String`, `ByStr32` or `ByStr20`. Values can be of any type.

- `m[k] := v`: In-place map insert key `k` and value `v` into `Map m`. If the intermediate key(s) does not exist in `Map m`, they are freshly created. To insert a value into a nested map, simply do `m[k1][k2][...] := v`.
- `delete m[k]`: In-place map removal of key `k`. If the intermediate key(s) does not exist, no action is taken. To delete a value in a nested map, simply do `delete m[k1][k2][...]`.
- `v <- m[k]`: In-place map fetch of value `v` from key `k`. Returns `Some value` after indexing with key(s). Returns `None` if key(s) does not exists. To fetch a value in a nested map, simply do `v <- m[k1][k2][...]`.
- `b <- exists m[k1][k2][...]`: In-place existence check to check if all keys have a value mapped. Returns `Bool`.

- `put m k v`: Insert key `k` and value `v` into `Map m`. Returns a new `Map` with the newly inserted key/value in addition to the key/value pairs contained earlier. This is typically used in library functions.
- `get m k`: In `Map m`, for key `k`, return the associated value as `Option v` (Check below for `Option` data type). The returned value is `None` if `k` is not in the map `m`. This is typically used in library functions.
- `remove m k`: Remove key `k` and its associated value from the map `m`. Returns a new updated `Map`. This is typically used in library functions.
- `contains m k`: Is key `k` and its associated value present in the map `m`. Returns `Bool`. This is typically used in library functions.
- `to_list m`: Convert `Map m` into a `List (Pair ('A) ('B))` where `'A` and `'B` are key and value types.

Addresses

Addresses are declared using the data type `ByStr20` data type. `ByStr20` literals being with `0x` and contain 20 bytes (40 hexadecimal characters).

The following operations on addresses are language built-in.

- `eq a1 a2`: Is `ByStr20` equal to `ByStr20`. Returns `Bool`.

Block Numbers

Block numbers have a dedicated type in Scilla. Variables of this type are specified with the keyword `BNum`. A `BNum` literal is a sequence of digits with the keyword `block` prefixed (example `block 101`).

The following `BNum` operations are language built-in.

- `eq b1 b2`: Is `BNum b1` equal to `BNum b2`. Returns `Bool`.
- `blt b1 b2`: Is `BNum b1` less than `BNum b2`. Returns `Bool`.
- `badd b1 i1`: Add `UintX i1` to `BNum b1`. Returns `BNum`.

3.4.3 Algebraic Data Types (ADTs)

Algebraic data types are composite types, used commonly in functional programming. The following ADTs are featured in Scilla. Each ADT is defined as a set of **constructors**. Each constructor takes a set of arguments of certain types.

Boolean

Boolean values are specified using the keyword `Bool`. `Bool` ADT has two constructors: `True` and `False` that do not take any argument. Thus the following code fragment constructs a `Bool` ADT that represents `True`:

```
x = True
```

Option

Similar to `Option` in OCaml, the `Option` ADT in Scilla provides means to represent the presence of a value `x` or the absence of any value. `Option` has two constructors `None` and `Some`.

- `Some` represents the presence of a value. `Some {`A} x` constructs an ADT that represents the presence of a value `x` of type `'A`. The following code fragment constructs an `Option` using the `Some` constructor with an argument of type `Int32`:

```
let x =
  let ten = Int32 10 in
  Some {Int32} 10
```

- `None` represents the absence of any value. `None {`A}` constructs an ADT that represents the absence of any value of type `'A`. The following code fragment constructs an `Option` using the `None` constructor with an argument of type `ByStr20`:

```
x = None {ByStr20}
```

They are extremely useful for initialising a mutable variable with no value.

```
field empty_bool : Option Bool : None {Bool}
```

Note that constructing `Some {(ADT)}` or `None {(ADT)}` will require the `( )` parentheses:

```
let one = Int32 1 in
x = Some {(Pair Int32 Int32)} one one
```

Some constructor is also frequently used in extracting values from a Map:

```
(*Assume m = Map ByStr20 Int32 that contains a key value pair of _
↪sender data*)
getValue = builtin get m _sender;
match getValue with
| Some v =>
  v = v + v;
  statements...
| None =>
  statements...
end
```

List

The `List` ADT, similar to Lists in other functional languages provides a structure to contain a list of values of the same type. A `List` is specified using the `List` keyword and has two constructors:

- `Nil` creates an empty `List`. It takes the following form: `Nil {'A}`, and creates an empty list of entries of type `'A`.
- `Cons` adds an element to an existing list. It takes the following form: `Cons {'A} h l`, where `'A` is a type variable that can be instantiated with any type and `h` is an element of type `'A` that is inserted at the head of list `l` (of type `List 'A`).

The following code example demonstrates building a list of `Int32` values. To do this, we start with an empty list `Nil {Int32}`. The rest of the list is built by inserting items into the list. The final list built in this example is `[11 -> 10 -> 2 -> 1 -> NIL]`.

```
let one = Int32 1 in
let two = Int32 2 in
let ten = Int32 10 in
```

(continues on next page)

(continued from previous page)

```
let eleven = Int32 11 in

let nil = Nil {Int32} in
let l1 = Cons {Int32} one nil in
let l2 = Cons {Int32} two l1 in
let l3 = Cons {Int32} ten l2 in
  Cons {Int32} eleven l3
```

The following two structural recursion primitives are provided for any `List`.

- `list_foldl: ('B -> 'A -> 'B) -> 'B -> (List 'A) -> 'B` : For any types `'A` and `'B`, `list_foldl` recursively processes the input list (`List 'A`) from left to right, by applying an iterator function (`'B -> 'A -> 'B`) to the element being processed and an accumulator (`'B`). The initial value of this accumulator is provided as argument to `list_foldl`.
- `list_foldr: ('A -> 'B -> 'B) -> 'B -> (List 'A) -> 'B` : Same as `list_foldl` but process the list elements from right to left.

To further illustrate `List` in Scilla, we show a small example using `list_foldl` to count the number of elements in a list.

```
1  let list_length =
2    tfun 'A =>
3    fun (l : List 'A) =>
4      let folder = @list_foldl 'A Int32 in
5      let init = Int32 0 in
6      let iter =
7        fun (h : 'A) =>
8        fun (z : Int32) =>
9          let one = Int32 1 in
10           builtin add one z
11       in
12         folder iter init l
```

`list_length` defines a function that takes one argument `l` of type `List 'A`, where `'A` is a parametric type (type variable), specified in `line 2`. We instantiate `list_foldl` in `line 4` for a list of type `'A` with the accumulator type being `Int32`. An initial value of `0` is used for the accumulator. The iterator function `iter` increments the accumulator as it is invoked by the folder for each element of the list `l`. The final value of the accumulator will be the number of increments or in other words, the number of elements in the list.

Common `List` utilities (including `list_length`) are provided in the `ListUtils` library, as part of the standard library distribution for Scilla.

Pair

`Pair` ADTs are used to contain a pair of values of possibly different types. `Pair` variables are specified using the `Pair` keyword and can be constructed using the constructor `Pair {'A 'B} a b` where `'A` and `'B` are type variables that can be instantiated to any type, and `a` and `b` are variables of type `'A` and `'B` respectively.

Below is an example to construct a `Pair` of `Int32` values.

```
let p =
  let one = 1 in
  let two = 2 in
  Pair {Int32 Int32} one two
  ...
```

Pair can be used to contain a pair of values with different types. For example, to declare a pair of types `String Uint32` and initialize it to a mutable field `pp`:

```
field pp: Pair (String) (Uint32) =
              let s1 = "Hello" in
              let num = Uint32 2 in
              Pair {(String) (Uint32)} s1 num
  ...
```

Note the difference in how we perform a type declaration `Pair{ (A') (B')}` and the syntax used to create a pair of values using the constructor `Pair (A') (B')`. In the type declaration, a pair of curly braces surounds the two data types `A'` and `B'`.

We now illustrate how pattern matching can be used to extract the first element from a `Pair`. The function `fst` shown below is defined in the `PairUtils` library of the Scilla standard library.

```
let fst =
  tfun 'A =>
  fun (p : Pair 'A 'A) =>
  match p with
  | Pair {'A 'A} a b =>
      a
  end

let p = Pair {Int32 Int32} one two in
let fst_int = @fst Int32 in
let a = fst_int p in
  ... (* a = one *) ...
```

Nat

Scilla provides an ADT to work with natural numbers. A natural number `Nat` is constructed using `Zero` or `Succ Nat`, i.e., the successor of a natural number. The following code shows the build up of `Nat` three:

```
let three =
  let zero = Zero in
  let one  = Succ zero in
  let two  = Succ one in
  Succ two
```

The following folding (structural recursion) is defined for `Nat` in Scilla, where `'T` is a parametric type variable.

```
nat_fold : ('T -> Nat -> 'T) -> 'T -> Nat -> 'T
```

Similar in spirit to the `List` folds described earlier, the `Nat` fold takes an initial accumulator (of type `'T`) and a function that takes as arguments a `Nat` and the intermediate accumulator (`'T`) and returns a new accumulator value. This iterator function has type `'T -> Nat -> 'T`. The fold iterates through all natural numbers, applying the iterator function and returns a final accumulator.

3.4.4 More ADT examples

To make it easier to understand how ADTs can be used, we provide two more examples and describe them in detail. Both the functions described below are distributed as `ListUtils` in the Scilla standard *library*.

List: Head

The code below extracts the first item of a `List` and returns it as an `Option`, i.e., `Some` element is returned if the list has at least one element, `None` otherwise. The given test case takes `[ 1 -> 2 -> 3 -> NIL]` as an input and returns `1`.

```
1  let list_head =
2    tfun 'A =>
3    fun (l : List 'A) =>
4      match l with
5      | Cons h t =>
6        Some h
7      | Nil =>
8        None
9      end
10 in
11
12 let int_head = @list_head Int32 in
13
14 let one = Int32 1 in
15 let two = Int32 2 in
16 let three = Int32 3 in
17 let nil = Nil {Int32} in
18
19 let l1 = Cons {Int32} three nil in
20 let l2 = Cons {Int32} two l1 in
21 let l3 = Cons {Int32} one l2 in
22 int_head l3
```

In `lines 14-21` we build a list that can be used as input to the `list_head` function. `Line 12` instantiates the `list_head` function for `Int32` and the last line invokes the instantiated `list_head` function.

`tfun 'A` in `line 2` specifies that `'A` is a parametric type / variable to the function, while `fun` in `line 3` specifies that `l` is a parameter of type `List 'A`. In other words, in `lines 1-3`, we are specifying a function `list_head` that can be instantiated for any type `'A` and takes as argument, a variable of type `List 'A`. The pattern matching in `line 5` matches for a `List` which is constructed as `Cons h t` where `h` is the head and `t` is the tail and returns the head as `Some h`. If the list is empty, then it matches the pattern match for `Nil` in `line 7` and returns `None`, indicating that the list has no head.

List: Exists

We now describe a function, which given a list and a predicate function, returns `True` if the predicate holds for at least one element of the list.

```
1  let list_exists =
2    tfun 'A =>
3    fun (f : 'A -> Bool) =>
4    fun (l : List 'A) =>
5      let folder = @list_foldl 'A Bool in
6      let init = False in
7      let iter =
8        fun (z : Bool) =>
9        fun (h : 'A) =>
10         let res = f h in
11         match res with
12         | True =>
```

(continues on next page)

(continued from previous page)

```
          True
        | False =>
          z
        end
    in
      folder iter init l

let int_exists = @list_exists Int128 in
let f =
  fun (a : Int128) =>
    let three = Int128 3 in
    builtin lt a three

...
(* build list l3 similar to previous example *)
...

(* check if l3 has at least one element satisfying f *)
int_exists f l3
```

Similar to the previous example, `'A` is a type variable to the function. The function takes two arguments (1) a list `l` of type `List 'A` and a predicate, i.e., a function that takes an element of the list (of type `'A`) and returns `True` or `False`, indicating satisfaction of the predicate.

To iterate through all elements of the input list `l`, we use `list_foldl`. An instantiation of `list_foldl` for list type `'A` and accummulator type `Bool` is done in `line 5`. The initial accummulator value is `False` (to indicate that no element that satisfies the predicate is seen yet). The iterator function `iter` defined in `line 6` tests the current list element provided as argument `h` for the predicate and returns an updated accummulator. If the accummulator is found `True` at some point, that value remains unchanged for the rest of the fold.

3.4.5 Standard Libraries

Scilla comes with four standard library contracts `BoolUtils.scilla`, `ListUtils.scilla`, `NatUtils.scilla` and `PairUtils.scilla`. As the name suggests these contracts respecively implement operations on `Bool`, `List`, `Nat` and `Pair` data types. In order to use the functions defined in these contracts, an `import` utility is provided. So, if one wants to use all the operations defined on `List`, one has to add `import ListUtils` just before the declaration of any contract-specific library, or add `import ListUtils PairUtils` if one wants to use operations in both libraries.

Below, we present the functions defined in each of the library.

BoolUtils

- `andb`: Computes the logical AND of two `Bool` values.
- `orb`: Computes the logical OR of two `Bool` values.
- `negb`: Computes the logical negation of a `Bool` value.

PairUtils

- `fst`: Extract the first element of a Pair.
- `snd`: Extract the second element of a Pair.

ListUtils

- `list_map : ('A -> 'B) -> List 'A -> : List 'B.`

  Apply `f : 'A -> 'B` to every element of `l : List 'A`.

```
(*Library*)
let f =
  func (a : Int32) =>
    sha256hash a

(*Contract transition*)
(*Assume l as a list [1 -> 2 -> 3 -> NIL]*)
transition
   hash_list_int32 = @list_map Int32;
   hashed_list = hash_list_int32 f l;
end
```

- `list_filter : ('A -> Bool) -> List 'A -> List 'A.`

  Preserving the order of elements in `l : List 'A`, return new list containing only those elements that satisfy the predicate `f : 'A -> Bool`. Linear complexity.

```
(*Library*)
let f =
  fun (a : Int32) =>
    let ten = Int32 10 in
    builtin lt a ten

(*Contract transition*)
(*Assume l as a list [1 -> 2 -> 3 -> 11 -> NIL]*)
transition
  less_ten_int32 = @list_filter Int32;
  less_ten_list = less_ten_int32 f l
  (*Returns a list [1 -> 2 -> 3 -> NIL]*)
end
```

- `list_head : (List 'A) -> (Option 'A).`

  Return the head element of a list `l : List 'A` as `Some 'A`, `None` if `l` is `Nil` (the empty list).

- `list_tail : (List 'A) -> (Option List 'A).`

  For input list `l : List 'A`, returns `Some l'`, where `l'` is `l` except for it's head; returns `Some Nil` if `l` has only one element; returns `None` if `l` is empty.

- `list_append : (List 'A -> List 'A -> List 'A).`

Append the second list to the first one and return a new List. Linear complexity (on first list).

- `list_reverse : (List 'A -> List 'A).`

  Return the reverse of the input list. Linear complexity.

- `list_flatten : (List List 'A) -> List 'A.`

  Concatenate a list of lists. Each element (`List 'A`) of the input (`List List 'A`) are all concatenated together (in the same order) to give the result. linear complexity over the total number of elements in all of the lists.

- `list_length : List 'A -> Int32`

  Number of elements in list. Linear complexity.

- `list_eq : ('A -> 'A -> Bool) -> List 'A -> List 'A -> Bool.`

  Takes a function `f : 'A -> 'A -> Bool` to compare elements of lists `l1 : List 'A` and `l2 : List 'A` and returns True if all elements of the lists compare equal. Linear complexity.

- `list_mem : ('A -> 'A -> Bool) -> 'A -> List 'A -> Bool.`

  Checks whether an element `a : 'A` is in the list `l : List'A`. `f : 'A -> 'A -> Bool` should be provided for equality comparison. Linear complexity.

```
(*Library*)
let f =
  fun (a : Int32) =>
  fun (b : Int32) =>
    builtin eq a b

(*transition*)
transition search (keynumber : Int32)

  (*Assume l is a list of Int32, say [1 -> 2 -> 3 -> 4 -> NIL]*)
  list_mem_int32 = @list_mem Int32;
  check_result = list_mem_int32 f keynumber l (*Return Bool*)

end
```

- `list_forall : ('A -> Bool) -> List 'A -> Bool.`

Return True if all elements of list `l : List 'A` satisfy predicate `f : 'A -> Bool`. Linear complexity.

- `list_exists : ('A -> Bool) -> List 'A -> Bool`.

  Return True if at least one element of list `l : List 'A` satisfies predicate `f : 'A -> Bool`. Linear complexity.

- `list_sort : ('A -> 'A -> Bool) -> List 'A -> List 'A`.

  Stable sort the input list `l : List 'A`. Function `flt : 'A -> 'A -> Bool` provided must return True if its first argument is lesser-than its second argument. Linear complexity.

```
(*Library*)
let int_sort = @list_sort Uint64 in

let flt =
  fun (a : Uint64) =>
  fun (b : Uint64) =>
    builtin lt a b

let zero = Uint64 0 in
let one = Uint64 1 in
let two = Uint64 2 in
let three = Uint64 3 in
let four = Uint64 4 in

(* l6 = 2 4 3 2 1 2 3 *)
  let l6 =
  let nil = Nil {Uint64} in
  let l0 = Cons {Uint64} two nil in
  let l1 = Cons {Uint64} four l0 in
  let l2 = Cons {Uint64} three l1 in
  let l3 = Cons {Uint64} two l2 in
  let l4 = Cons {Uint64} one l3 in
  let l5 = Cons {Uint64} two l4 in
  Cons {Uint64} three l5

(*transition*)
transition sortList ()

  (* res1 = 1 2 2 2 3 3 4 *)
  res1 = int_sort flt l6

end
```

- `list_find : ('A -> Bool) -> 'A -> 'A`.

  Return `Some a`, where `a` is the first element of `l : List 'A` that satisfies the predicate `f : 'A -> Bool`. Returns `None` if none of the elements in `l` satisfy `f`. Linear complexity.

- `list_zip : List 'A -> List 'B -> List (Pair 'A 'B).`

  Combine corresponding elements of `m1 : List 'A` and `m2 : List 'B` into a `Pair` and return the resulting list. In case of different number of elements in the lists, the extra elements are ignored.

- `list_zip_with : ('A -> 'B -> 'C) -> List 'A -> List 'B -> List 'C ).` Linear complexity.

  Combine corresponding elements of `m1 : List 'A` and `m2 : List 'B` using `f : 'A -> 'B -> 'C` and return the resulting list of `'C`. In case of different number of elements in the lists, the extra elements are ignored.

- `list_unzip : List (Pair 'A 'B) -> Pair (List 'A) (List 'B).`

  Convert a list `l : Pair 'A 'B` of `Pair`s into a `Pair` of lists. Linear complexity.

- `list_nth : Int32 -> List 'A -> Option 'A.`

  Returns `Some 'A` if n'th element exists in list. `None` otherwise. Linear complexity.

```
(*transition*)
(*Assume l as a list of Int32 [1 -> 2 -> 3 -> NIL]*)
transition search_nth (nth : Int32)
  list_nth_int32 = @list_nth Int32;
  search_nth = list_nth_int32 nth l;
  match search_nth with
  | Some v =>
    statements...
  | None =>
    statements...
  end
end
```

3.5 Interpreter Interface

The Scilla interpreter executable provides a calling interface that enables users to invoke transitions with specified inputs and obtain outputs. Execution of a contract with supplied inputs will result in a set of outputs and a change in the smart contract mutable state.

3.5.1 Calling Interface

A transition defined in a smart contract can be called either by the issuance of a transaction or by message calls from another smart contract. The same calling interface will be used to call the contract via external transactions and inter-contract message calls.

The inputs to the interpreter (`scilla-runner`) consists of four input JSON files as described below. Every invocation of the interpreter to execute a transition must be provided with these four JSON inputs:

```
./scilla-runner -init init.json -istate input_state.json -iblockchain input_
↪blockchain.json -imessage input_message.json -o output.json -i input.scilla
```

The interpreter executable can be run either to create a contract (denoted `CreateContract`) or to invoke a transition (function) in a contract (`InvokeContract`). Depending on which of these two, some of the arguments will be absent. The table below outlays the arguments that should be present in each of these two cases. A `CreateContract` is distinguished from an `InvokeContract`, based on the presence of `input_message.json` and `input_state.json`. If these arguments are absent, then the interpreter will evaluate it as a `CreateContract`. Else, it will treat it as an `InvokeContract`. Note that for `CreateContract`, the interpreter only performs basic checks such as matching the contract's immutable parameters with `init.json` and whether the contract definition is free of syntax errors.

		Present	
Input	Description	`CreateContract`	`InvokeContract`
`init.json`	Immutable contract params	Yes	Yes
`input_state.json`	Mutable contract state	No	Yes
`input_blockchain.json`	Blockchain state	Yes	Yes
`input_message.json`	Transition and params	No	Yes
`output.json`	Output	Yes	Yes
`input.scilla`	Input contract	Yes	Yes

3.5.2 Initializing the Immutable State

`init.json` defines the values of the immutable paramters of a contract. It does not change between invocations. The JSON is an array of objects, each of which contains the following fields:

Field	Description
vname	Name of the immutable contract parameter
type	Type of the immutable contract parameter
value	Value of the immutable contract parameter

Example 1

For the `HelloWorld.scilla` contract fragment given below, we have only one immutable variable `owner`.

```
contract HelloWorld
 (* Immutable parameters *)
 (owner: ByStr20)
```

A sample `init.json` for this contract will look like the following:

```
[
    {
        "vname" : "_scilla_version",
        "type" : "Uint32",
        "value" : "0"
    },
    {
        "vname" : "owner",
```

(continues on next page)

(continued from previous page)

```
        "type" : "ByStr20",
        "value" : "0x1234567890123456789012345678901234567890"
    },
    {
        "vname" : "_creation_block",
        "type" : "BNum",
        "value" : "1"
    }
]
```

Example 2

For the `Crowdfunding.scilla` contract fragment given below, we have three immutable variables `owner`, `max_block` and `goal`.

```
contract Crowdfunding
    (* Immutable parameters *)
    (owner     : ByStr20,
     max_block : BNum,
     goal      : UInt128)
```

A sample `init.json` for this contract will look like the following:

```
[
  {
      "vname" : "_scilla_version",
      "type" : "Uint32",
      "value" : "0"
  },
  {
      "vname" : "owner",
      "type" : "ByStr20",
      "value" : "0x1234567890123456789012345678901234567890"
  },
  {
      "vname" : "max_block",
      "type" : "BNum" ,
      "value" : "199"
  },
  {
      "vname" : "goal",
      "type" : "Uint128",
      "value" : "500"
  },
  {
      "vname" : "_creation_block",
      "type" : "BNum",
      "value" : "1"
  }
]
```

3.5.3 Input Blockchain State

`input_blockchain.json` feeds the current blockchain state to the interpreter. It is similar to `init.json`, except that it is a fixed size array of objects, where each object has `vname` fields only from a pre-determined set (which correspond to actual blockchain state variables).

Permitted JSON fields: Only JSONs that differ in the `value` field as per the example below are permitted currently.

```
[
    {
        "vname" : "BLOCKNUMBER",
        "type"  : "BNum",
        "value" : "3265"
    }
]
```

3.5.4 Input Message

`input_message.json` contains the required information to invoke a transition. The json is an array containing the following four objects:

Field	Description
_tag	Transition to be invoked
_amount	Number of ZILs to be transferred
_sender	Address of the invoker
params	An array of parameter objects

All the four fields are mandatory. `params` can be empty if the transition takes no parameters.

The `params` array is encoded similar to how `init.json` is encoded, with each parameter specifying the (`vname`, `type`, `value`) that has to be passed to the transition that is being invoked.

Example 1

For the following transition:

```
transition SayHello()
```

an example `input_message.json` is given below:

```
{
    "_tag"    : "SayHello",
    "_amount" : "0",
    "_sender" : "0x1234567890123456789012345678901234567890",
    "params"  : []
}
```

Example 2

For the following transition:

```
transition TransferFrom (from : ByStr20, to : ByStr20, tokens : Uint128)
```

an example `input_message.json` is given below:

```
{
  "_tag"    : "TransferFrom",
  "_amount" : "0",
  "_sender" : "0x6434567890123456789012345678901234567890cd",
  "params"  : [
    {
      "vname" : "from",
      "type"  : "ByStr20",
      "value" : "0x1234567890123456789012345678901234567890"
    },
    {
      "vname" : "to",
      "type"  : "ByStr20",
      "value" : "0x7834567890123456789012345678901234567890cd"
    },
    {
      "vname" : "tokens",
      "type"  : "Uint128",
      "value" : "500"
    }
  ]
}
```

3.5.5 Interpreter Output

The interpreter will return a JSON object (`output.json`) with the following fields:

Field	Description
scilla_major_version	The major version of the scilla language of this contract.
gas_remaining	The remaining gas after invoking or deploying a contract.
_accepted	Whether the contract has accepted ZIL (Either `true`/`false`)
message	The emitted message to another contract/non-contract account.
states	An array of objects that form the new contract state

- `message` is a JSON object that will have a similar format to `input_message.json`, except that instead of `_sender` field, it will have a `_recipient` field. The fields in `message` are given below:

Field	Description
_tag	Transition to be invoked
_amount	Number of ZILs to be transferred
_recipient	Address of the recipient
params	An array of parameter objects to be passed

  The `params` array is encoded similar to how `init.json` is encoded, with each parameter specifying the (`vname`, `type`, `value`) that has to be passed to the transition that is being invoked.

- `states` is an array of objects that represents the mutable state of the contract. Each entry of the `states` array also specifies (`vname`, `type`, `value`).

Example 1

The example below is an output generated by `HelloWorld.scilla`.

```
{
  "scilla_major_version": "0",
  "gas_remaining": "7402",
  "_accepted": "false",
  "message": {
    "_tag": "Main",
    "_amount": "0",
    "_recipient": "0x1234567890123456789012345678901234567890",
    "params": [
      {
        "vname": "code",
        "type": "Int32",
        "value": "2"
      }
    ]
  },
  "states": [
    {
      "vname": "_balance",
      "type": "Uint128",
      "value": "0"
    },
    {
      "vname": "welcome_msg",
      "type": "String",
      "value": "Hello World"
    }
  ],
  "events": []
}
```


Example 2

Another slightly more involved example with `Map` in `states`.


```
{
  "scilla_major_version": "0",
  "gas_remaining": "7359",
  "_accepted": "true",
  "message": null,
  "states": [
    {
      "vname": "_balance",
      "type": "Uint128",
      "value": "100"
    },
    {
      "vname": "backers",
      "type": "Map (ByStr20) (Uint128)",
      "value": [
        {
          "key": "0x123456789012345678901234567890123456789ab",
          "val": "100"
        }
      ]
    },
```


(continues on next page)

(continued from previous page)

```
    {
      "vname": "funded",
      "type": "Bool",
      "value": { "constructor": "False", "argtypes": [], "arguments": [] }
    }
  ],
  "events": [
    {
      "_eventname": "DonationSuccess",
      "params": [
        {
          "vname": "donor",
          "type": "ByStr20",
          "value": "0x1234567890123456789012345678901234567890ab"
        },
        {
          "vname": "amount",
          "type": "Uint128",
          "value": "100"
        },
        { "vname": "code",
          "type": "Int32",
          "value": "1"
        }
      ]
    }
  ]
}
```

Note: For mutable variables of type `Map`, the first entry in the `value` field are the types of the `key` and `value`. Also, note that the `value` field of a variable of type `ADT` has several fields namely, `constructor`, `argtypes` and `arguments`.

3.5.6 Input Mutable Contract State

`input_state.json` contains the current value of mutable state variables. It has the same forms as the `states` field in `output.json`. An example of `input_state.json` for `Crowdfunding.scilla` is given below.

```
[
  {
    "vname": "backers",
    "type": "Map (ByStr20) (Uint128)",
    "value": [
      {
        "key": "0x1234567890123456789012345678901234567890cd",
        "val": "200"
      },
      {
        "key": "0x1234567890123456789012345678901234567890ab",
        "val": "100"
      }
    ]
  },
```

(continues on next page)

(continued from previous page)

```
  {
    "vname": "funded",
    "type": "Bool",
    "value": {
      "constructor": "False",
      "argtypes": [],
      "arguments": []
    }
  },
  {
    "vname": "_balance",
    "type": "Uint128",
    "value": "300"
  }
]
```

3.6 Contact

Questions? Ask us on our Gitter Channel.