scilla-doc Documentation

Release 0.5.0

Amrit Kumar

Dec 05, 2019

Contents

Scilla (short for *Smart Contract Intermediate-Level LAnguage*) is an intermediate-level smart contract language being developed for the Zilliqa blockchain. Scilla is 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.

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 in distinguishing 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. You can try out Scilla in the online IDE.

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 Example Contracts

Savant IDE comes 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.
- **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*.
- **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()`.

- **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()`.
- **FungibleToken** : Fungible token contract that mimics an ERC20 style fungible token standard. De facto 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.
- **ZilGame** : 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.
- **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 a 20 byte address).
- 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 a 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 = ""
```

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 using a transition that takes a previous state and an input and yields a new state. Check the 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 =>
```

(continues on next page)

(continued from previous page)

```
5      e = {_eventname : "setHello"; code : not_owner_code};
6      event e
7    | True =>
8      welcome_msg := msg;
9      e = {_eventname : "setHello"; code : set_hello_code};
10     event e
11   end
12 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 using *pattern matching*, the syntax of which is given in the fragment below.

```
match expr with
| pattern_1 => expr_1
| pattern_2 => expr_2
end
```

The above code checks whether `expr` evaluates to a value that matches `pattern_1` or `pattern_2`. If `expr` evaluates to a value matching `pattern_1`, then the next expression to be evaluated will be `expr_1`. Otherwise, if `expr` evaluates to a value matching `pattern_2`, then the next expression to be evaluated will be `expr_2`.

Hence, the following code block implements an `if-then-else` instruction:

```
match expr with
| True  => expr_1
| False => expr_2
end
```

The Caller is Not the Owner

In case the caller is different from `owner`, the transition takes the `False` branch and the contract emits an event using the instruction `event`.

An event is a signal that gets stored on the blockchain for everyone to see. If a user uses a client application to invoke a transition on a contract, the client application can listen for events that the contract may emit, and alert the user.

More concretely, the output event in this case is:

```
e = {_eventname : "setHello"; code : not_owner_code};
```

An event is comprised of a number of `vname : value` pairs delimited by `;` inside a pair of curly braces `{}`. An event must contain the compulsory field `_eventname`, and may contain other fields such as the `code` field in the example above.

Note: In our example we have chosen to name the event after the transition that emits the event, but any name can be chosen. However, it is recommended that you name the events in a way that makes it easy to see which part of the

code emitted the event.

The Caller is the 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`. This is done through the following instruction:

```
welcome_msg := msg;
```

Note: Writing to a mutable variable is done using the operator `:=`.

And as in the previous case, the contract then emits an event with the code `set_hello_code`.

Libraries

A Scilla contract may come with some helper libraries that declare purely functional components of a contract, i.e., components with no state manipulation. 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 as follows:

```
library HelloWorld
```

The library may include utility functions and program constants using the `let x = y in expr` construct. In our example the library will only include the definition of error codes:

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

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

```
library HelloWorld

 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 =>
     e = {_eventname : "setHello"; code : not_owner_code};
     event e
   | True =>
     welcome_msg := msg;
     e = {_eventname : "setHello"; code : set_hello_code};
     event e
   end
 end
```

Adding Another Transition

We may now add the second transition `getHello()` that allows client applications to know what the `welcome_msg` is. The declaration is similar to `setHello (msg : String)` except that `getHello()` does not take a parameter.

```
transition getHello ()
    r <- welcome_msg;
    e = {_eventname: "getHello"; msg: r};
    event e
end
```

Note: Reading from a contract state variable is done using the operator `<-`.

In the `getHello()` transition, we will first read from a mutable variable, and then we construct and emit the event.

Scilla Version

Once a contract has been deployed on the network, it cannot be changed. It is therefore necessary to specify which version of Scilla the contract is written in, so as to ensure that the behaviour of the contract does not change even if changes are made to the Scilla specification.

The Scilla version of the contract is declared using the keyword `scilla_version`:

```
scilla_version 0
```

The version declaration must appear before any library or contract code.

Putting it All Together

The complete contract that implements the desired specification is given below, where we have added comments using the `(* *)` construct:

```
(* HelloWorld contract *)

(***************************************************)
(*                 Scilla version                  *)
(***************************************************)

scilla_version 0

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

let not_owner_code  = Uint32 1
let set_hello_code  = Uint32 2

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

contract HelloWorld
```

(continues on next page)

(continued from previous page)

```
(owner: ByStr20)

field welcome_msg : String = ""

transition setHello (msg : String)
  is_owner = builtin eq owner _sender;
  match is_owner with
  | False =>
    e = {_eventname : "setHello"; code : not_owner_code};
    event e
  | True =>
    welcome_msg := msg;
    e = {_eventname : "setHello"; code : set_hello_code};
    event e
  end
end

transition getHello ()
  r <- welcome_msg;
  e = {_eventname: "getHello"; msg: r};
  event e
end
```

3.3.2 A Second Example: 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 until a certain, pre-determined block number is reached on the blockchain (`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 immutable variables are provided when the contract is deployed. At that point we wish to add a sanity check that the `goal` is a strictly positive amount. If the contract is accidentally initialised with a `goal` of 0, then the contract should not be deployed.

The total amount that has been donated to the campaign so far is stored in a field `_balance`. Any contract in Scilla has an implicit `_balance` field of type `Uint128`, which is initialised to 0 when the contract is deployed, and which holds the amount of ZIL in the contract's account on the blockchain.

The campaign is deemed successful if the owner can raise the 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.

Sanity check for contract parameters

To ensure that the `goal` is a strictly positive amount, we use a contract constraint:

```
with
  let zero = Uint128 0 in
  builtin lt zero goal
=>
```

This ensures that the contract cannot be deployed with a `goal` of 0 by mistake.

Reading the Current Block Number

The deadline is given as a block number, so to check whether the deadline has passed, we must compare the deadline against the current block number.

The current block number is read as follows:

```
blk <- & BLOCKNUMBER;
```

Block numbers have a dedicated type `BNum` in Scilla, so as to not confuse them with regular unsigned integers.

Note: Reading data from the blockchain is done using the operator `<- &`. Blockchain data cannot be updated directly from the contract.

Reading and Updating the Current Balance

The target for the campaign is specified by the owner in the immutable variable `goal` when the contract is deployed. To check whether the target have been met, we must compare the total amount raised to the target.

The amount of ZIL raised is stored in the contract's account on the blockchain, and can be accessed through the implicitly declared `_balance` field as follows:

```
bal <- _balance;
```

Money is represented as values of type `Uint128`.

Note: The `_balance` field is read using the operator `<-` just like any other contract state variable. However, the `_balance` field can only be updated by accepting money from incoming messages (using the instruction `accept`), or by explicitly transferring money to other account (using the instruction `send` as explained below).

Sending Messages

In Scilla, there are two ways that transitions can transmit data. One way is through events, as covered in the previous example. The other is through the sending of messages using the instruction `send`.

`send` is used to send messages to other accounts, either in order to invoke transitions on another smart contract, or to transfer money to user accounts. On the other hand, events are dispatched signals that smart contracts can use to transmit data to client applications.

To construct a message we use a similar syntax as when constructing events:

```
msg = {_tag : ""; _recipient : owner; _amount : bal; code : got_funds_code};
```

A message must contain the compulsory fields _tag, _recipient and _amount. The _recipient field is the blockchain address (of type ByStr20) that the message is to be sent to, and the _amount field is the number of ZIL to be transferred to that account.

The value of the _tag field is the name of the transition (of type String) that is to be invoked on the _recipient contract. If _recipient is a user account, then the value of _tag can be set to be "" (the empty string). In fact, if the _recipient is a user account, then the value of _tag is ignored.

In addition to the compulsory fields the message may contain other fields, such as code above. However, if the message recipient is a contract, the additional fields must have the same names and types as the parameters of the transition being invoked on the recipient contract.

Sending a message is done using the send instruction, which takes a list of messages as a parameter. Since we will only ever send one message at a time in the crowdfunding contract, we define a library function one_msg to construct a list consisting of one message:

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

To send out a message, we first construct the message, insert it into a list, and send it:

```
msg = {_tag : ""; _recipient : owner; _amount : bal; code : got_funds_code};
msgs = one_msg msg;
send msgs
```

Note: The Zilliqa blockchain does not yet support sending multiple messages in the same transition. This means that the list given as an argument to send must contain only one message, and that a transition may perform at most one send instruction each time the transition is called.

Procedures

The transitions of a Scilla contract often need to perform the same small sequence of instructions. In order to prevent code duplication a contract may define a number of *procedures*, which may be invoked from the contract's transitions. Procedures also help divide the contract code into separate, self-contained pieces which are easier to read and reason about individually.

A procedure is declared using the keyword procedure. The end of a procedure is declared using the keyword end. The procedure keyword is followed by the transition name, then the input parameters within (), and then the statements of the procedure.

In our example the Donate transition will issue an event in three situations: An error event if the donation happens after the deadline, another error event if the backer has donated money previously, and a non-error event indicating a successful donation. Since much of the event issuing code is identical, we decide to define a procedure DonationEvent which is responsible for issuing the correct event:

```
procedure DonationEvent (failure : Bool, error_code : Int32)
  match failure with
  | False =>
    e = {_eventname : "DonationSuccess"; donor : _sender;
         amount : _amount; code : accepted_code};
    event e
  | True =>
    e = {_eventname : "DonationFailure"; donor : _sender;
```

(continues on next page)

(continued from previous page)

```
        amount : _amount; code : error_code};
    event e
  end
end
```

The procedure takes two arguments: A `Bool` indicating whether the donation failed, and an error code indicating the type of failure if a failure did indeed occur.

The procedure performs a `match` on the `failure` argument. If the donation did not fail, the error code is ignored, and a `DonationSuccess` event is issued. Otherwise, if the donation failed, then a `DonationFailure` event is issued with the error code that was passed as the second argument to the procedure.

The following code shows how to invoke the `DonationEvent` procedure with the arguments `True` and `0`:

```
c = True;
err_code = Int32 0;
DonationEvent c err_code;
```

Note: The special variables `_sender` and `_amount` are available to the procedure even though the procedure is invoked by a transition rather than by an incoming message. It is not necessary to pass the variables as arguments to the procedure.

Note: Procedures are similar to library functions in that they can be invoked from any transition (as long as the transition is defined after the procedure). However, procedures are different from library functions in that library functions cannot access the contract state, and procedures cannot return a value.

Procedures are similar to transitions in that they can access and change the contract state, as well as read the incoming messages and send outgoing messages. However, procedures cannot be invoked from the blockchain layer. Only transitions may be invoked from outside the contract, so procedures can be viewed as private transitions.

Putting it All Together

The complete crowdfunding contract is given below.

```
(***************************************************)
(*                 Scilla version                  *)
(***************************************************)

scilla_version 0

(***************************************************)
(*               Associated library                *)
(***************************************************)
import BoolUtils

library Crowdfunding

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

(continues on next page)

(continued from previous page)

```
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 get_funds_allowed =
  fun (cur_block : BNum) =>
  fun (max_block : BNum) =>
  fun (balance : Uint128) =>
  fun (goal : Uint128) =>
    let in_time = blk_leq cur_block max_block in
    let deadline_passed = negb in_time in
    let target_not_reached = builtin lt balance goal in
    let target_reached = negb target_not_reached in
    andb deadline_passed target_reached

let claimback_allowed =
  fun (balance : Uint128) =>
  fun (goal : Uint128) =>
  fun (already_funded : Bool) =>
    let target_not_reached = builtin lt balance goal in
    let not_already_funded = negb already_funded in
    andb target_not_reached not_already_funded

let accepted_code = Int32 1
let missed_deadline_code = Int32 2
let already_backed_code  = Int32 3
let not_owner_code  = Int32 4
let too_early_code  = Int32 5
let got_funds_code  = Int32 6
let cannot_get_funds  = Int32 7
let cannot_reclaim_code = Int32 8
let reclaimed_code = Int32 9

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

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

(* Contract constraint *)
with
  let zero = Uint128 0 in
  builtin lt zero goal
=>

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

procedure DonationEvent (failure : Bool, error_code : Int32)
```

(continues on next page)

(continued from previous page)

```
  match failure with
  | False =>
    e = {_eventname : "DonationSuccess"; donor : _sender;
         amount : _amount; code : accepted_code};
    event e
  | True =>
    e = {_eventname : "DonationFailure"; donor : _sender;
         amount : _amount; code : error_code};
    event e
  end
end

procedure PerformDonate ()
  c <- exists backers[_sender];
  match c with
  | False =>
    accept;
    backers[_sender] := _amount;
    DonationEvent c accepted_code
  | True =>
    DonationEvent c already_backed_code
  end
end

transition Donate ()
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    PerformDonate
  | False =>
    t = True;
    DonationEvent t missed_deadline_code
  end
end

procedure GetFundsFailure (error_code : Int32)
  e = {_eventname : "GetFundsFailure"; caller : _sender;
       amount : _amount; code : error_code};
  event e
end

procedure PerformGetFunds ()
  bal <- _balance;
  tt = True;
  funded := tt;
  msg = {_tag : ""; _recipient : owner; _amount : bal; code : got_funds_code};
  msgs = one_msg msg;
  send msgs
end

transition GetFunds ()
  is_owner = builtin eq owner _sender;
  match is_owner with
  | False =>
    GetFundsFailure not_owner_code
  | True =>
```

(continues on next page)

(continued from previous page)

```
    blk <- & BLOCKNUMBER;
    bal <- _balance;
    allowed = get_funds_allowed blk max_block bal goal;
    match allowed with
    | False =>
      GetFundsFailure cannot_get_funds
    | True =>
      PerformGetFunds
    end
  end
end

procedure ClaimBackFailure (error_code : Int32)
  e = {_eventname : "ClaimBackFailure"; caller : _sender;
       amount : _amount; code : error_code};
  event e
end

procedure PerformClaimBack (amount : Uint128)
  delete backers[_sender];
  msg = {_tag : ""; _recipient : _sender; _amount : amount; code : reclaimed_code};
  msgs = one_msg msg;
  e = { _eventname : "ClaimBackSuccess"; caller : _sender; amount : amount; code :␣
↪reclaimed_code};
  event e;
  send msgs
end

transition ClaimBack ()
  blk <- & BLOCKNUMBER;
  after_deadline = builtin blt max_block blk;
  match after_deadline with
  | False =>
    ClaimBackFailure too_early_code
  | True =>
    bal <- _balance;
    f <- funded;
    allowed = claimback_allowed bal goal f;
    match allowed with
    | False =>
      ClaimBackFailure cannot_reclaim_code
    | True =>
      res <- backers[_sender];
      match res with
      | None =>
        (* Sender has not donated *)
        ClaimBackFailure cannot_reclaim_code
      | Some v =>
        PerformClaimBack v
      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:

- The contract starts with the declaration of `scilla_version`, which indicates which major Scilla version the contract uses.
- Then follows the declaration of a `library` that contains purely mathematical functions, e.g., a function to compute the boolean `AND` of two bits, or a function computing the factorial of a given natural number.
- Then follows the actual contract definition declared using the keyword `contract`.
- Within a contract, there are then four distinct parts:
  1. The first part declares the immutable parameters of the contract.
  2. The second part describes the contract's constraint, which must be valid when the contract is deployed.
  3. The third part declares the mutable fields.
  4. The fourth part contains all `transition` and `procedure` definitions.

```
(* Scilla contract structure *)

(***************************************************)
(*                 Scilla version                  *)
(***************************************************)

scilla_version 1

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

library MyContractLib


(* Library code block follows *)



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

contract MyContract

(* Immutable fields declaration *)

(vname_1 : vtype_1,
 vname_2 : vtype_2)

(* Contract constraint *)
with
  (* Constraint expression *)
=>
```

(continues on next page)

(continued from previous page)

```
(* Mutable fields declaration *)

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

(* Transitions and procedures *)


(* Procedure signature *)
procedure firstProcedure (param_1 : type_1, param_2 : type_2)
  (* Procedure body *)

end

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

end

(* Procedure signature *)
procedure secondProcedure (param_1 : type_1, param_2 : type_2)
  (* Procedure 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 when the contract is deployed, and cannot be modified afterwards.

Immutable variables are declared using the following syntax:

```
(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 when the contract is deployed.

Note: In addition to the explicitly declared immutable fields, a Scilla contract has an implicitly declared mutable field `_this_address` of type `ByStr20`, which is initialised to the address of the contract when the contract is deployed. This field can be freely read within the implementation, but cannot be modified.

Contract Constraints

A *contract constraint* is a requirement placed on the the contract's initial parameters. A contract constraint provides a way of establishing a contract invariant as soon as the contract is deployed, thus preventing the contract being deployed

with non-sensical parameters.

A contract constraint is declared using the following syntax:

```
with
  ...
=>
```

The constraint must be an expression of type `Bool`.

The constraint is checked when the contract is deployed. Contract deployment only succeeds if the constraint evaluates to `True`. If it evaluates to `False`, then the deployment fails.

Note: Declaring a constract constraint is optional. If no constraint is declared, then the constraint is assumed to simply be `True`.

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 the field in question. The definitions complete the initial state of the contract, at the time of creation. As the contract executes a transition, the values of these fields get modified.

Note: In addition to the explicitly declared mutable fields, a Scilla contract has an implicitly declared mutable field `_balance` of type `Uint128`, which is initialised to 0 when the contract is deployed. The `_balance` field 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 (using `send`), or by accepting money from incoming messages (using `accept`).

Note: Both mutable and immutable variables must be of a *storable* type:

- Messages, events and the special `Unit` type are not storable. All other primitive types like integers and strings are storable.
- Function types are not storable.
- Complex types involving uninstantiated type variables are not storable.
- Maps and ADT are storable if the types of their subvalues are storable. For maps this means that the key type and the value type must both be storable, and for ADTs this means that the type of every constructor argument must be storable.

Units

The Zilliqa protocol supports three basic tokens units - ZIL, LI (10^-6 ZIL) and QA (10^-12 ZIL).

The base unit used in Scilla smart contracts is QA. Hence, when using money variables, it is important to attach the trailing zeroes that are needed to represent it in QAs.

```
(* fee is 1 QA *)
let fee = Uint128 1

(* fee is 1 LI *)
let fee = Uint128 1000000

(* fee is 1 ZIL *)
let fee = Uint128 1000000000000
```

Transitions

Transitions are a way to define now the state of the contract may change. The transitions of a contract define the public interface for the contract, since transitions may be invoked by sending a message to the contract.

Transitions 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 the parameters that are explicitly declared in the definition, each transition has the following implicit parameters:

- `_sender : ByStr20` : The account address that triggered this transition. If the transition was called by a contract account instead of a user account, then `_sender` is the address of the contract that called this transition.
- `_amount : Uint128` : Incoming amount of QAs (see section above on the units) sent by the sender. To transfer the money from the sender to the contract, the transition must explicitly accept the money using the `accept` instruction. The money transfer does not happen if the transition does not execute an `accept`.

Note: Transition parameters must be of a *serialisable* type:

- Messages, events and the special `Unit` type are not serialisable. All other primitive types like integers and strings are serialisable.
- Function types and map types are not serialisable.
- Complex types involving uninstantiated type variables are not serialisable.
- ADT are serialisable if the types of their subvalues are serialisable. This means that the type of every constructor argument must be serialisable.

Procedures

Procedures are another way to define now the state of the contract may change, but in contrast to transitions, procedures are not part of the public interface of the contract, and may not be invoked by sending a message to the contract. The only way to invoke a procedure is to call it from a transition or from another procedure.

Procedures are defined with the keyword `procedure` followed by the parameters to be passed. The definition ends with the `end` keyword.

```
procedure 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 `,`.

Once a procedure is defined it is available to invoked from transitions and procedures in the rest of the contract file. It is not possible to invoke a procedure from transition or procedure defined earlier in the contract, nor is it possible for a procedure to call itself recursively.

Procedures are invoked using the name of the procedure followed by the actual arguments to the procedure:

```
v1 = ...;
v2 = ...;
foo v1 v2;
```

All arguments must be supplied when the procedure is invoked. A procedure does not return a result.

Note: The implicit transition parameters `_sender` and `_amount` are implicitly passed to all the procedures that a transition calls. There is therefore no need to declare those parameters explicitly when defining a procedure.

Note: Procedure parameters cannot be (or contain) maps. If a procedure needs to access a map, it is therefore necessary to either make the procedure directly access the contract field containing the map, or use a library function to perform the necessary computations on the map.

Expressions

Expressions handle pure operations. Scilla contains the following types of expressions:

- `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 or event 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.
- `f x` : Apply the function `f` to the parameter `x`.

- `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`: Apply the type function `x` to the type `T`.
- `builtin f x`: Apply the built-in function `f` on `x`.
- `match` expression: Matches a bound variable with patterns and evaluates the expression in that clause. The `match` expression is similar to the `match` expression in OCaml. The pattern to be matched can be an ADT constructor (see *ADTs*) with subpatterns, a variable, or a wildcard _. An ADT constructor pattern matches values constructed with the same constructor if the subpatterns match the corresponding subvalues. A variable matches anything, and binds the variable to the value it matches in the expression of that clause. A wildcard matches anything, but the value is then ignored.

```
match x with
| pattern_1 =>
  expression_1 ...
| pattern_2 =>
  expression_2 ...
| _ => (*Wildcard*)
  expression ...
end
```

Note: A pattern-match must be exhaustive, i.e., every legal (type-safe) value of `x` must be matched by a pattern. Additionally, every pattern must be reachable, i.e., for each pattern there must be a legal (type-safe) value of `x` that matches that pattern, and which does not match any pattern preceding it.

Statements

Statements in Scilla are operations with effect, and hence not purely mathematical. Scilla contains the following types of statements:

- `x <- f` : Fetch the value of the contract field `f`, and store it into the local variable `x`.
- `f := x` : Update the mutable contract field `f` with the value of `x`. `x` may be a local variable, or another contract field.
- `x <- & BLOCKNUMBER` : Fetch the value of the blockchain state variable `BLOCKNUMBER`, and store it into the local variable `x`.
- `v = e` : Evaluate the expression `e`, and assign the value to the local variable `v`.
- `p x y z` : Invoke the procedure `p` with the arguments `x`, `y` and `z`. The number of arguments supplied must correspond to the number of arguments the procedure takes.
- `match` : Pattern-matching at statement level:

```
match x with
| pattern_1 =>
  statement_11;
  statement_12;
  ...
| pattern_2 =>
  statement_21;
  statement_22;
  ...
```

(continues on next page)

(continued from previous page)

```
| _ => (*Wildcard*)
  statement_n1;
  statement_n2;
  ...
end
```

- `accept` : Accept the QAs of the message that invoked the transition. The amount is automatically added to the `_balance` field of the contract. If a message contains QAs, but the invoked transition does not accept the money, the money is transferred back to the sender of the message.
- `send` and `event` : Communication with the blockchain. See the next section for details.
- In-place map operations : Operations on contract fields of type `Map`. See the *Maps* section for details.

A sequence of statements must be separated by semicolons `;` :

```
transition T ()
  statement_1;
  statement_2;
  ...
  statement_n
end
```

Notice that the final statement does not have a trailing `;`, since `;` is used to separate statements rather than terminate them.

Communication

A contract can communicate with other contract and user accounts through the `send` instruction:

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

  The following code snippet defines a `msg` with four entries `_tag`, `_recipient`, `_amount` and `param`.

```
(*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 };
```

A message passed to `send` must contain the compulsory fields `_tag`, `_recipient` and `_amount`.

The `_recipient` field (of type `ByStr20`) is the blockchain address that the message is to be sent to, and the `_amount` field (of type `Uint128`) is the number of ZIL to be transferred to that account.

The `_tag` field (of type `String`) is only used when the value of the `_recipient` field is the address of a contract. In this case, the value of the `_tag` field is the name of the transition that is to be invoked on the recipient contract. If the recipient is a user account, the `_tag` field is ignored.

In addition to the compulsory fields the message may contain other fields (of any type), such as `param` above. However, if the message recipient is a contract, the additional fields must have the same names and types as the parameters of the transition being invoked on the recipient contract.

Here's an example that sends multiple messages.

```
msg1 = { _tag : "setFoo"; _recipient : contractAddress1; _amount : Uint128 0;
↪ foo : Uint32 101 };
msg2 = { _tag : "setBar"; _recipient : contractAddress2; _amount : Uint128 0;
↪ bar : Uint32 100 };
```

(continues on next page)

(continued from previous page)

```
msgs =
  let nil = Nil {Message} in
  let m1 = Cons {Message} msg1 nil in
  Cons msg2 m1
  ;
send msgs
```

A contract can also communicate to the outside world by emitting events. An event is a signal that gets stored on the blockchain for everyone to see. If a user uses a client application invoke a transition on a contract, the client application can listen for events that the contract may emit, and alert the user.

- `event e`: Emit a message `e` as an event. The following code emits an event with name `e_name`.

```
e = { _eventname : "e_name"; <entry>_2 ; <entry>_3 };
event e
```

An emitted event must contain the compulsory field `_eventname` (of type `String`), and may contain other entries as well. The value of the `_eventname` entry must be a string literal. All events with the same name must have the same entry names and types.

Note: A transition may send a message at any point during execution (including during the execution of the procedures it invokes), but the recipient account will not receive the message until after the transition has completed. Similarly, a transition may emit events at any point during execution (including during the execution of the procedures it invokes), but the event will not be visible on the blockchain before the transition has completed.

Run-time Errors

A contract can raise errors by throwing exceptions. Any error in the execution of a transition (including those due to thrown exceptions, out-of-gas errors and others such as integer overflows) results in the blockchain aborting the execution of the contract as well as aborting any other contracts that were executed before in that chain.

The syntax for raising errors is similar to that of events and messages.

```
e = { _exception : "InvalidInput"; <entry>_2; <entry>_3 };
throw e
```

Unlike that for `event` or `send`, The argument to `throw` is optional and can be omitted. An empty throw will result in an error that just conveys the location of where the `throw` happened without more information.

Note: We do not currently support catching exceptions and may add this in the future.

Gas consumption in Scilla

Deploying contracts and executing transitions in them cost gas. The detailed cost mechanism is explained here.

The Nucleus Wallet page can be used to estimate gas costs for some transactions .

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, the type of an unsigned integer of 32 bits is `Uint32`.

The following code snippet declares a variable of type `Uint32`:

```
let x = Uint32 43
```

Scilla supports the following built-in operations on integers. Each operation takes two integers `IntX`/`UintX` (of the same type) as arguments, except for `pow` whose second argument is always `Uint32`

- `builtin eq i1 i2` : Is `i1` equal to `i2`? Returns a `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`: The remainder of integer division of `i1` by `i2`. Returns an integer of the same type.
- `builtin lt i1 i2`: Is `i1` less than `i2`? Returns a `Bool`.
- `builtin pow i1 i2`: `i1` raised to the power of `i2`. Returns an integer of the same type as `i1`.
- `builtin to_nat i1`: Convert a value of type `Uint32` to the equivalent value of type `Nat`.
- `builtin to_(u)int(32/64/128/256)`: Convert a `UintX/IntX` or `String` (that represents a number) value to the equivalent `UintX/IntX` value. Returns `Some IntX/UintX` if the conversion succeeded, `None` otherwise.

Addition, subtraction, multiplication, pow, division and remainder operations may raise integer overflow, underflow and division_by_zero errors.

Note: Variables related to blockchain money, such as the `_amount` entry of a message or the `_balance` field of a contract, are of type `Uint128`.

Strings

`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 variable of type `String`:

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

Scilla supports the following built-in operations on strings:

- `builtin eq s1 s2` : Is `s1` equal to `s2`? Returns a `Bool`. `s1` and `s2` must be of type `String`.
- `builtin concat s1 s2` : Concatenate string `s1` with string `s2`. Returns a `String`.
- `builtin substr s idx len` : Extract the substring of `s` of length `len` starting from position `idx`. `idx` and `len` must be of type `Uint32`. Character indices in strings start from `0`. Returns a `String` or fails with a runtime error if the combination of the input parameters results in an invalid substring.

- `builtin to_string x`: Convert `x` to a string literal. Valid types of `x` are `IntX`, `UintX`, `ByStrX` and `ByStr`. Returns a `String`.
- `builtin strlen s` : Calculate the length of `s` (of type `String`). Returns a `Uint32`.

Crypto Built-ins

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

The following code snippet declares a variable of type `ByStr32`:

```
let x = 0x1234567890123456789012345678901234567890123456789012345678901234567890abff
```

Scilla supports the following built-in operations on hashes and other cryptographic primitives, including byte sequences. In the description below, `Any` can be of type `IntX`, `UintX`, `String`, `ByStr20` or `ByStr32`.

- `builtin eq h1 h2`: Is `h1` equal to `h2`? Returns a `Bool`.
- `builtin sha256hash x` : Convert `x` of `Any` type to its SHA256 hash. Returns a `ByStr32`.
- `builtin keccak256hash x`: Convert `x` of `Any` type to its Keccak256 hash. Returns a `ByStr32`.
- `builtin ripemd160hash x`: Convert `x` of `Any` type to its RIPEMD-160 hash. Returns a `ByStr20`.
- `builtin to_bystr h` : Convert a hash `h` of type `ByStrX` (for some known `X`) to one of arbitrary length of type `ByStr`.
- `builtin to_uint256 h` : Convert a hash `h` to the equivalent value of type `Uint256`. `h` must be of type `ByStrX` for some known `X` less than or equal to 32.
- `builtin schnorr_verify pubk data sig` : Verify a signature `sig` of type `ByStr64` against a byte string `data` of type `ByStr` with the Schnorr public key `pubk` of type `ByStr33`.
- `builtin ecdsa_verify pubk data sig` : Verify a signature `sig` of type `ByStr64` against a byte string `data` of type `ByStr` with the ECDSA public key `pubk` of type `ByStr33`.
- `concat h1 h2`: Concatenate the hashes `h1` and `h2`. If `h1` has type `ByStrX` and `h2` has type `ByStrY`, then the result will have type `ByStr(X+Y)`.
- `builtin bech32_to_bystr20 prefix addr`. The builtin takes a network specific prefix (`"zil"` / `"tzil"`) of type `String` and an input bech32 string (of type `String`) and if the inputs are valid, converts it to a raw byte address (*ByStr20*). The return type is `Option ByStr20`. On success, `Some addr` is returned and on invalid inputs `None` is returned.
- `builtin bystr20_to_bech32 prefix addr`. The builtin takes a network specific prefix (`"zil"` / `"tzil"`) of type `String` and an input `ByStr20` address, and if the inputs are valid, converts it to a bech32 address. The return type is `Option String`. On success, `Some addr` is returned and on invalid inputs `None` is returned.
- `builtin alt_bn128_G1_add p1 p2`. The builtin takes two points `p1`, `p2` on the `alt_bn128` curve and returns the sum of the points in the underlying group G1. The input points and the result point are each a `Pair {Bystr32 ByStr32}`. Each scalar component `ByStr32` of a point is a big-endian encoded number. Also see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md
- `builtin alt_bn128_G1_mul p s`. The builtin takes a point `p` on the `alt_bn128` curve (as desribed previously), and a scalar `ByStr32` value `s` and returns the sum of the point `p` taken `s` times. The result is a point on the curve.
- `builtin alt_bn128_pairing_product pairs`. This builtin takes in a list of pairs `pairs` of points. Each pair consists of a point in group G1 (`Pair {Bystr32 ByStr32}`) as the first component and a point

in group G2 (`Pair {Bystr64 ByStr64}`) as the second component. Hence the argument has type `List {(Pair (Pair ByStr32 ByStr32) (Pair ByStr64 ByStr64)) }`. The function applies a pairing function on each point to check for equality and returns `True` or `False` depending on whether the pairing check succeeds or fails. Also see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md

Maps

A value of type `Map kt vt` provides a key-value store where `kt` is the type of keys and `vt` is the type of values. `kt` may be any one of `String`, `IntX`, `UintX`, `ByStrX` or `ByStr`. `vt` may be any type except a function type.

Scilla supports the following built-in operations on maps:

- `builtin put m k v`: Insert a key `k` bound to a value `v` into a map `m`. Returns a new map which is a copy of the `m` but with `k` associated with `v`. If `m` already contains key `k`, the old value bound to `k` gets replaced by `v` in the result map. The value of `m` is unchanged. The `put` function is typically used in library functions. Note that `put` makes a copy of `m` before inserting the key-value pair.
- `m[k] := v`: *In-place* insert operation, i.e., identical to `put`, but without making a copy of `m`. `m` must refer to a contract field. Insertion into nested maps is supported with the syntax `m[k1][k2][...] := v`. If the intermediate key(s) does not exist in the nested maps, they are freshly created along with the map values they are associated with.
- `builtin get m k`: Fetch the value associated with the key `k` in the map `m`. Returns an optional value (see the `Option` type below) – if `k` has an associated value `v` in `m`, then the result is `Some v`, otherwise the result is `None`. The `get` function is typically used in library functions.
- `v <- m[k]`: *In-place* fetch operation, i.e, identical to `get`. `m` must refer to a contract field. Returns an optional value (see the `Option` type below) – if `k` has an associated value `v` in `m`, then the result is `Some v`, otherwise the result is `None`. Fetching from nested maps is supported with the syntax `v <- m[k1][k2][...]`. If one or more of the intermediate key(s) do not exist in the corresponding map, the result is `None`.
- `builtin contains m k`: Is the key `k` associated with a value in the map `m`? Returns a `Bool`. The `contains` function is typically used in library functions.
- `b <- exists m[k]`: *In-place* existence check, i.e., identical to `contains`. `m` must refer to a contract field. Returns a `Bool`. Existence checks through nested maps is supported with the syntax `v <- exists m[k1][k2][...]`. If one or more of the intermediate key(s) do not exist in the corresponding map, the result is `False`.
- `builtin remove m k`: Remove a key `k` and its associated value from the map `m`. Returns a new map which is a copy of `m` but with `k` being unassociated with a value. The value of `m` is unchanged. If `m` does not contain key `k` the `remove` function simply returns a copy of `m` with no indication that `k` is missing. The `remove` function is typically used in library functions. Note that `remove` makes a copy of `m` before removing the key-value pair.
- `delete m[k]`: *In-place* remove operation, i.e., identical to `remove`, but without making a copy of `m`. `m` must refer to a contract field. Removal from nested maps is supported with the syntax `delete m[k1][k2][...]`. If any of the specified keys do not exist in the corresponding map, no action is taken. Note that in the case of a nested removal `delete m[k1][...][kn-1][kn]`, only the key-value association of `kn` is removed. The key-value bindings of `k` to `kn-1` will still exist.
- `builtin to_list m`: Convert a map `m` to a `List (Pair kt vt)` where `kt` and `vt` are key and value types, respectively (see the `List` type below).
- `builtin size m`: Return the number of bindings in map `m`. The result type is `Uint32`.

Note: Builtin functions `put` and `remove` return a new map, which is a possibly modified copy of the original map. This may affect performance!

Addresses

An address in Scilla is declared using the data type `ByStr20`. `ByStr20` represents a hexadecimal byte string of 20 bytes (40 hexadecimal characters). A `ByStr20` literal is prefixed with `0x`.

Scilla supports the following built-in operations on addresses:

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

Block Numbers

Block numbers have a dedicated type `BNum` in Scilla. Variables of this type are specified with the keyword `BNum` followed by an integer value (for example `BNum 101`).

Scilla supports the following built-in operations on block numbers:

- `eq b1 b2`: Is `b1` equal to `b2`? Returns a `Bool`.
- `blt b1 b2`: Is `b1` less than `b2`? Returns a `Bool`.
- `badd b1 i1`: Add `i1` of type `UintX` to `b1` of type `BNum`. Returns a `BNum`.
- `bsub b1 b2`: Subtract `b2` from `b1`, both of type `BNum`. Returns an `Int256`.

3.4.3 Algebraic Datatypes

An *algebraic datatype* (ADT) is a composite type used commonly in functional programming. Each ADT is defined as a set of **constructors**. Each constructor takes a set of arguments of certain types.

Scilla is equipped with a number of built-in ADTs, which are described below. Additionally, Scilla allows users to define their own ADTs.

Boolean

Boolean values are specified using the type `Bool`. The `Bool` ADT has two constructors `True` and `False`, neither of which take any arguments. Thus the following code fragment constructs a value of type `Bool` by using the constructor `True`:

```
x = True
```

Option

Optional values are specified using the type `Option t`, where `t` is some type. The `Option` ADT has two constructors:

- `Some` represents the presence of a value. The `Some` constructor takes one argument (the value, of type `t`).
- `None` represents the absence of a value. The `None` constructor takes no arguments.

The following code snippet constructs two optional values. The first value is an absent string value, constructed using `None`. The second value is the `Int32` value 10, which, because the value is present, is constructed using `Some`:

```
let none_value = None {String}

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

Optional values are useful for initialising fields where the value is not yet known:

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

Optional values are also useful for functions that might not have a result, such as the `get` function for maps:

```
getValue = builtin get m _sender;
match getValue with
| Some v =>
  (* _sender was associated with v in m *)
  v = v + v;
  ...
| None =>
  (* _sender was not associated with a value in m *)
  ...
end
```

List

Lists of values are specified using the type `List t`, where `t` is some type. The `List` ADT has two constructors:

- `Nil` represents an empty list. The `Nil` constructor takes no arguments.
- `Cons` represents a non-empty list. The `Cons` constructor takes two arguments: The first element of the list (of type `t`), and another list (of type `List t`) representing the rest of the list.

All elements in a list must be of the same type `t`. In other words, two values of different types cannot be added to the same list.

The following example shows how to build a list of `Int32` values. First we create an empty list using the `Nil` constructor. We then add four other values one by one using the `Cons` constructor. Notice how the list is constructed backwards by adding the last element, then the second-to-last element, and so on, so that the final list is `[11; 10; 2; 1]`:

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

Scilla provides three structural recursion primitives for lists, which can be used to traverse all the elements of any list:

- `list_foldl: ('B -> 'A -> 'B) -> 'B -> (List 'A) -> 'B` : Recursively process the elements in a list from front to back, while keeping track of an *accumulator* (which can be thought of as a running total). `list_foldl` takes three arguments, which all depend on the two type variables `'A` and `'B`:
  - The function processing the elements. This function takes two arguments. The first argument is the current value of the accumulator (of type `'B`). The second argument is the next list element to be processed (of type `'A`). The result of the function is the next value of the accumulator (of type `'B`).
  - The initial value of the accumulator (of type `'B`).
  - The list of elements to be processed (of type `(List 'A)`).

The result of applying `list_foldl` is the value of the accumulator (of type `'B`) when all list elements have been processed.

- `list_foldr: ('A -> 'B -> 'B) -> 'B -> (List 'A) -> 'B` : Similar to `list_foldl`, except the list elements are processed from back to front. Notice also that the processing function takes the list element and the accumulator in the opposite order from the order in `list_foldl`.
- `list_foldk: ('B -> 'A -> ('B -> 'B) -> 'B) -> 'B -> (List 'A) -> 'B` : Recursively process the elements in a list according to a *folding function*, while keeping track of an *accumulator*. `list_foldk` is a more general version of the left and right folds, which, by the way, can be both implemented in terms of it. `list_foldk` takes three arguments, which all depend on the two type variables `'A` and `'B`:
  - The function describing the fold step. This function takes three arguments. The first argument is the current value of the accumulator (of type `'B`). The second argument is the next list element to be processed (of type `'A`). The third argument represents the postponed recursive call (of type `'B -> 'B`). The result of the function is the next value of the accumulator (of type `'B`). The computation *terminates* if the programmer does not invoke the postponed recursive call. This is a major difference between `list_foldk` and the left and right folds which process their input lists from the beginning to the end unconditionally.
  - The initial value of the accumulator `z` (of type `'B`).
  - The list of elements to be processed (of type `List 'A`).

Note: When an ADT takes type arguments (such as `List 'A`), and occurs inside a bigger type (such as the type of `list_foldl`), the ADT and its arguments must be grouped using parentheses `( )`. This is the case even when the ADT occurs as the only argument to another ADT. For instance, when constructing an empty list of optional values of type `Int32`, one must instantiate the list type using the syntax `Nil {(Option Int32)}`.

To further illustrate the `List` type in Scilla, we show a small example using `list_foldl` to count the number of elements in a list. For an example of `list_foldk` see *list_find*.

```
1  let list_length : forall 'A. List 'A -> Uint32 =
2    tfun 'A =>
3    fun (l : List 'A) =>
4    let foldl = @list_foldl 'A Uint32 in
5    let init = Uint32 0 in
6    let one = Uint32 1 in
7    let iter =
8      fun (z : Uint32) =>
9      fun (h : 'A) =>
10       builtin add one z
11   in
12     foldl iter init l
```

`list_length` defines a function that takes a type argument `'A`, and a normal (value) argument `l` of type `List 'A`.

`'A` is a *type variable* which must be instantiated by the code that intends to use `list_length`. The type variable is specified in line 2.

In line 4 we instantiate the types for `list_foldl`. Since we are traversing a list of values of type `'A`, we pass `'A` as the first type argument to `list_foldl`, and since we are calculating the length of the list (a non-negative integer), we pass `Uint32` as the accumulator type.

In line 5 we define the initial value of the accumulator. Since an empty list has length 0, the initial value of the accumulator is 0 (of type `Uint32`, to match the accumulator type).

In lines 6-10 we specify the processing function `iter`, which takes the current accumulator value `z` and the current list element `h`. In this case the processing function ignores the list element, and increments the accumulator by 1.

When all elements in the list have been processed, the accumulator will have been incremented as many times as there are elements in the list, and hence the final value of the accumulator will be equal to the length of the list.

In line 12 we apply the type-instantiated version of `list_foldl` from line 4 to the processing function, the initial accumulator, and the list of values.

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

Pair

Pairs of values are specified using the type `Pair t1 t2`, where `t1` and `t2` are types. The `Pair` ADT has one constructor:

- `Pair` represents a pair of values. The `Pair` constructor takes two arguments, namely the two values of the pair, of types `t1` and `t2`, respectively.

Note: `Pair` is both the name of a type and the name of a constructor of that type. An ADT and a constructor typically only share their names when the constructor is the only constructor of the ADT.

A `Pair` value may contain values of different types. In other words, `t1` and `t2` need not be the same type.

Below is an example where we declare a field `pp` of type `Pair String Uint32`, which we then initialise by constructing a pair consisting of a value of type `String` and a value of type `Uint32`:

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

Notice the difference in how we specify the type of the field as `Pair A' B'`, and how we specify the types of values given to the constructor as `Pair { A' 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 =>
  tfun 'B =>
  fun (p : Pair ('A) ('B)) =>
    match p with
    | Pair a b => a
    end
```

Note: Using `Pair` is generally discouraged. Instead, the programmer should define an ADT which is specialised to the particular type of pairs that is needed in the particular use case. See the section on *User-defined ADTs* below.

Nat

Peano numbers are specified using the type `Nat`. The `Nat` ADT has two constructors:

- `Zero` represents the number 0. The `Zero` constructor takes no arguments.

- `Succ` represents the successor of another Peano number. The `Succ` constructor takes one argument (of type `Nat`) which represents the Peano number that is one less than the current number.

The following code shows how to build the Peano number corresponding to the integer 3:

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

Scilla provides two structural recursion primitives for Peano numbers, which can be used to traverse all the Peano numbers from a given `Nat` down to `Zero`:

- `nat_fold: ('A -> Nat -> 'A) -> 'A -> Nat -> 'A`: Recursively process the succession of numbers from a `Nat` down to `Zero`, while keeping track of an accumulator. `nat_fold` takes three arguments, two of which depend on the type variable `'A`:
  - The function processing the numbers. This function takes two arguments. The first argument is the current value of the accumulator (of type `'A`). The second argument is the next Peano number to be processed (of type `Nat`). Incidentally, the next number to be processed is the predecessor of the current number being processed. The result of the function is the next value of the accumulator (of type `'A`).
  - The initial value of the accumulator (of type `'A`).
  - The first Peano number to be processed (of type `Nat`).

  The result of applying `nat_fold` is the value of the accumulator (of type `'A`) when all Peano numbers down to `Zero` have been processed.
- `nat_foldk: ('A -> Nat -> ('A -> 'A) -> 'A) -> 'A -> Nat -> 'A`: Recursively process the Peano numbers down to zero according to a *folding function*, while keeping track of an *accumulator*. `nat_foldk` is a more general version of the left fold allowing for early termination. It takes three arguments, two depending on the type variable `'A`.
  - The function describing the fold step. This function takes three arguments. The first argument is the current value of the accumulator (of type `'A`). The second argument is the predecessor of the Peano number being processed (of type `Nat`). The third argument represents the postponed recursive call (of type `'A -> 'A`). The result of the function is the next value of the accumulator (of type `'A`). The computation *terminates* if the programmer does not invoke the postponed recursive call. Left folds inevitably process the whole list whereas `nat_foldk` can differ in this regard.
  - The initial value of the accumulator `z` (of type `'A`).
  - The Peano number to be processed (of type `Nat`).

To better understand `nat_foldk`, we explain how `nat_eq` works. `nat_eq` checks to see if two Peano numbers are equivalent. Below is the program, with line numbers and an explanation.

```
1  let nat_eq : Nat -> Nat -> Bool =
2  fun (n : Nat) => fun (m : Nat) =>
3    let foldk = @nat_foldk Nat in
4    let iter =
5      fun (n : Nat) => fun (ignore : Nat) => fun (recurse : Nat -> Nat) =>
6        match n with
7        | Succ n_pred => recurse n_pred
8        | Zero => m   (* m is not zero in this context *)
9        end in
10   let remaining = foldk iter n m in
11   match remaining with
```

(continues on next page)

(continued from previous page)

```
12    | Zero => True
13    |   _ => False
14    end
```

Line 2 specifies that we take two Peano numbers `m` and `n`. Line 3 instantiates the type of `nat_foldk`, we give it `Nat` because we will be passing a `Nat` value as the fold accumulator.

Lines 4 to 8 specify the fold description, this is the first argument that `nat_foldk` takes usually of type `'A -> Nat -> ('A -> 'A) -> 'A` but we have specified that `'A` is `Nat` in this case. Our function takes the accumulator `n` and `ignore : Nat` is the predecessor of the number being processed which we don't care about in this particular case.

Essentially, we start accumulating the end result from `n` and iterate at most `m` times (see line 10), decrementing both `n` and `m` at each recursive step (lines 4 - 9). The `m` variable gets decremented implicitly because this is how `nat_foldk` works under the hood. And we explicitly decrement `n` using pattern matching (lines 6, 7). To continue iteratively decrement both `m` and `n` we use `recurse` on line 7. If the two input numbers are equal, we will get the accumulator (`n`) equal to zero in the end. We call the final value of the accumulator `remaining` on line 10. At the end we will be checking to see if our accumulator ended up at `Zero` to say if the input numbers are equal. The last lines, return `True` when the result of the fold is `Zero` and `False` otherwise as described above.

In the case when accumulator `n` reaches zero (line 8) while `m` still has not been fully processed, we stop iteration (hence no `recurse` on that line) and return a non-zero natural number to indicate inequality. Any number (e.g. `Succ Zero`) would do, but to make the code concise we return the original input number `m` because we know `iter` gets called on `m` only if it's not zero.

In the symmetrical case when `m` reaches zero while the accumulator `n` is still strictly positive, we indicate inequality, because `remaining` gets this final value of `n`.

User-defined ADTs

In addition to the built-in ADTs described above, Scilla supports user-defined ADTs.

ADT definitions may only occur in the library parts of a program, either in the library part of the contract, or in an imported library. An ADT definition is in scope in the entire library in which it is defined, except that an ADT definition may only refer to other ADT definitions defined earlier in the same library, or in imported libraries. In particular, an ADT definition may not refer to itself in an inductive/recursive manner.

Each ADT defines a set of constructors. Each constructor specifies a number of types which corresponds to the number and types of arguments that the constructor takes. A constructor may be specified as taking no arguments.

The ADTs of a contract must have distinct names, and the set of all constructors of all ADTs in a contract must also have distinct names. Both the ADT and constructor names must begin with a capital letter ('A' - 'Z'). However, a constructor and an ADT may have the same name, as is the case with the `Pair` type whose only constructor is also called `Pair`.

As an example of user-defined ADTs, consider the following type declarations from a contract implementing a chess-like game called Shogi or Japanese Chess (https://en.wikipedia.org/wiki/Shogi). When in turn, a player can choose to either move one of his pieces, place a previously captured piece back onto the board, or resign and award the victory to the opponent.

The pieces of the game can be defined using the following type `Piece`:

```
type Piece =
| King
| GoldGeneral
| SilverGeneral
```

(continues on next page)

(continued from previous page)

```
| Knight
| Lance
| Pawn
| Rook
| Bishop
```

Each of the constructors represents a type of piece in the game. None of the constructors take any arguments.

The board is represented as a set of squares, where each square has two coordinates:

```
type Square =
| Square of Uint32 Uint32
```

The type `Square` is an example of a type where a constructor has the same name as the type. This usually happens when a type has only one constructor. The constructor `Square` takes two arguments, both of type `Uint32`, which are the coordinates (the row and the column) of the square on the board.

Similar to the definition of the type `Piece`, we can define the type of direction of movement using a constructor for each of the legal directions as follows:

```
type Direction =
| East
| SouthEast
| South
| SouthWest
| West
| NorthWest
| North
| NorthEast
```

We are now in a position to define the type of possible actions that a user may choose to perform when in turn:

```
type Action =
| Move of Square Direction Uint32 Bool
| Place of Piece Square
| Resign
```

If a player chooses to move a piece, she should use the constructor `Move`, and provide four arguments:

- An argument of type `Square`, indicating the current position of the piece she wants to move.
- An argument of type `Direction`, indicating the direction of movement.
- An argument of type `Uint32`, indicating the distance the piece should move.
- An argument of type `Bool`, indicating whether the moved piece should be promoted after being moved.

If instead the player chooses to place a previously captured piece back onto the board, she should use the constructor `Place`, and provide two arguments:

- An argument of type `Piece`, indicating which piece to place on the board.
- An argument of type `Square`, indicating the position the piece should be placed in.

Finally, if the player chooses to resign and award the victory to her opponent, she should use the constructor `Resign`. Since `Resign` does not take any arguments, no arguments should be provided.

To check which action a player has chosen we use a match statement or a match expression:

```
transition PlayerAction (action : Action)
  ...
  match action with
  | Resign =>
    ...
  | Place piece square =>
    ...
  | Move square direction distance promote =>
    ...
  end;
  ...
end
```

3.4.4 More ADT examples

To further illustrate how ADTs can be used, we provide some more examples and describe them in detail. Versions of both the functions described below can be found in the `ListUtils` part of the Scilla standard *library*.

Computing the Head of a List

The function `list_head` returns the first element of a list.

Since a list may be empty, `list_head` may not always be able to compute a result, and thus should return a value of the `Option` type. If the list is non-empty, and the first element is `h`, then `list_head` should return `Some h`. Otherwise, if the list is empty, `list_head` should return `None`.

The following code snippet shows the implementation of `list_head`, and how to apply it:

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

Line 2 specifies that `'A` is a type parameter to the function, while line 3 specifies that `l` is a (value) parameter of type `List 'A`. In other words, lines 1-3 specify a function `list_head` which can be instantiated for any type `'A`, and which takes as an argument a value of type `List 'A`.

The pattern-match in lines 4-9 matches on the value of `l`. In line 5 we match on the list constructor `Cons h t`, where `h` is the first element of the list, and `t` is the rest of the list. If the list is not empty then the match is successful, and

we return the first element as an optional value `Some h`. In line 7 we match on the list constructor `Nil`. If the list is empty then the match is successful, and we return the optional value `None` indicating that there was no head element of the list.

Line 11 instantiates the `list_head` function for the type `Int32`, so that `list_head` can be applied to values of type `List Int32`. Lines 13-20 build a list of type `List Int32`, and line 21 invokes the instantiated `list_head` function on the list that was built.

Computing a Left Fold

The function `list_foldl` returns the result of a left fold given a function `f : 'B -> 'A -> 'B`, accumulator `z : 'B` and list `xs : List 'A`. This can be implemented as a recursion primitive or a list utility function.

A left fold is a recursive application of an accumulator `z` and next list element `x : 'A` with `f` repetitively until there are no more list elements. For example the left fold on `[1,2,3]` using subtraction starting with accumulator 0 would be `((0-1)-2)-3 = -6`. The left fold is explained in pseudocode below, note that the result is always the accumulator type.

```
1 list_foldl _ z [] = z
2 list_foldl f z (x:xs) = list_foldl f (f z x) xs
```

The same can be achieved with `list_foldk` by partially applying a left fold description; this avoids illegal direct recursion. Our fold description `left_f : 'B -> 'A -> ('B -> 'B) -> 'B` takes arguments accumulator, next list element and recursive call. The recursive call will be supplied by the `list_foldk` function. An implementation is explained below.

```
1 let list_foldl : forall 'A. forall 'B. ( 'B -> 'A -> 'B) -> 'B -> List 'A -> 'B =
2 tfun 'A => tfun 'B =>
3 fun (f : 'B -> 'A -> 'B) =>
4 let left_f = fun (z: 'B) => fun (x: 'A) =>
5   fun (recurse : 'B -> 'B) => let res = f z x in
6   recurse res in
7 let folder = @list_foldk 'A 'B in
8 folder left_f
```

On line 1, we declare the name and type signature as according to the first paragraph. On the second line, we say that the function takes two types as arguments `'A` and `'B`. The third line says that we take some function `f` to process the list element and accumulator, as in paragraph two.

On line 4, we define the fold description using `f`. The fold description does not take a function but instead it should be implemented in terms of some function, as according to the type signature, `left_f : 'B -> 'A -> ('B -> 'B) -> 'B`. `left_f` takes arguments as described in paragraph two. We calculate the new accumulator `f z x` and call it `res`. Then we recursively call with the new accumulator.

On line 7, we instantiate an instance of `list_foldk` that has the right types for the job using a type application.

On line 8, we partially apply `folder` with the left fold description. . What is significant about `list_foldk` is that when calling the description, it provides a recursive call to itself, changing to the next element in the list and respective tail each time. This results in a function that just needs the user to provide the updated accumulator in the description.

Computing a Right Fold

The function `list_foldr` returns the result of a right fold given some function `f : 'A -> 'B -> 'B`, accumulator `z : 'B` and list `xs : List 'A`. Like `list_foldl`, this can be a recursion primitive or a list utility function.

A right fold is similar to a left fold but is reversed in a way. The right fold applies a function `f` with an accumulator `z` starting from the end and then combines with the second last element, third last element, etc... until it reaches the beginning. For example a right fold on the list `[1,2,3]` with subtraction starting with accumulator 0 would be equal to `1-(2-(3-0)) = 2`. It is listed below in pseudocode, note that the result is always the accumulator type.

```
1 list_foldr _ z [] = z
2 list_foldr f z (x:xs) = f x (list_foldr f z xs)
```

Like before, the same can be achieved with `list_foldk` by partially applying a right fold description. The fold description takes arguments accumulator `z : 'B`, next list element `x : 'A` and recursive call `recurse : 'B -> 'B`. The recursive call will be supplied by the `list_foldk` function. An implementation is explained below.

```
1 let list_foldr : forall 'A. forall 'B. ('A -> 'B -> 'B) -> 'B -> List 'A -> 'B =
2 tfun 'A => tfun 'B =>
3 fun (f : 'A -> 'B -> 'B) =>
4 let right_f = fun (z: 'B) => fun (x: 'A) =>
5   fun (recurse : 'B -> 'B) => let res = recurse z in f x res in
6 let folder = @list_foldk 'A 'B in
7 folder right_f
```

This is very similar to before. On line 1 we declare the name and type signature, according to the first paragraph. On line 2, we take two type arguments `'A` and `'B`. The third line says that we take some function `f` to process the list element `x : 'A` and accumulator `z`. The argument order is necessarily different to that of a left fold.

Following that we write a fold description like before. `list_foldk` processes lists from left to right. But we need `list_foldr` to emulate the right-to-left traversal. By calling `recurse z` on line 5 as our first action, we postpone actual computation with the combining function `f` preserving the original accumulator until the very end. Once the recursive call reaches an empty list it returns the original accumulator. Then the function calls `f x res` (line 5) will evaluate outwards combining from the end to the beginning, see paragraph two.

The recursive call `recurse z` on line 5 may seem to be the same each time but what is changing is the list element we process.

On line 6, we instantiate `list_foldk` by applying the types `'A` and `'B` to make a type-specific function. The last line we partially apply `folder` with the right fold description. Like before what is special about `list_foldk` is that it calls this function with a recursive call to itself that each time slightly truncates the list; this provides the recursion.

Checking for Existence in a List

The function `list_exists` takes a predicate function and a list, and returns a value indicating whether the predicate holds for at least one element in the list.

A predicate function is a function returning a boolean value, and since we want to apply it to elements in the list, the argument type of the function should be the same as the element type of the list.

`list_exists` should return either `True` (if the predicate holds for at least one element) or `False` (if the predicate does not hold for any element in the list), so the return type of `list_exists` should be `Bool`.

The following code snippet shows the implementation of `list_exists`, and how to apply it:

```
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
```

(continues on next page)

(continued from previous page)

```
7       let iter =
8         fun (z : Bool) =>
9         fun (h : 'A) =>
10          let res = f h in
11          match res with
12          | True =>
13            True
14          | False =>
15            z
16          end
17      in
18        folder iter init l
19
20  let int_exists = @list_exists Int128 in
21  let f =
22    fun (a : Int128) =>
23      let three = Int128 3 in
24      builtin lt a three
25
26  (* build list l3 similar to the previous example *)
27  ...
28
29  (* check if l3 has at least one element satisfying f *)
30  int_exists f l3
```

As in the previous example `'A` is a type variable to the function. The function takes two arguments:

- A predicate `f`, i.e., a function that returns a `Bool`. In this case, `f` will be applied to elements of the list, so the argument type of the predicate should be `'A`. Hence, `f` should have the type `'A -> Bool`.
- A list of elements `l` of type `List 'A`, so that the type of the elements in the list matches the argument type of `f`.

To traverse the elements of the input list `l` we use `list_foldl`. In line 5 we instantiate `list_foldl` for lists with elements of type `'A` and for the accumulator type `Bool`. In line 6 we set the initial accumulator value to `False` to indicate that no element satisfying the predicate has yet been seen.

The processing function `iter` defined in lines 7-16 tests the predicate on the current list element, and returns an updated accumulator. If an element has been found which satisfies the predicate, the accumulator is set to `True` and remains so for the rest of the traversal.

The final value of the accumulator is either `True`, indicating that `f` returned `True` for at least one element in the list, or `False`, indicating that `f` returned `False` for all elements in the list.

In line 20 we instantiate `list_exists` to work on lists of type `Int128`. In lines 21-24 we define the predicate, which returns `True` if its argument is less than 3, and returns `False` otherwise.

Omitted in line 27 is building the same list `l3` as in the previous example. In line 30 we apply the instantiated `list_exists` to the predicate and the list.

Finding the first occurrence satisfying a predicate

The function `list_find` searches for the first occurrence in a list that satisfies some predicate `p : 'A -> Bool`. It takes the predicate and the list, returning `Some {'A} x :: Option 'A` if `x` is the first element such that `p x` and `None {'A} :: Option 'A` otherwise.

Below we have an implementation of `list_find` that illustrates how to use `list_foldk`.

```
1  let list_find : forall 'A. ('A -> Bool) -> List 'A -> Option 'A =
2  tfun 'A =>
3  fun (p : 'A -> Bool) =>
4    let foldk = @list_foldk 'A (Option 'A) in
5    let init = None {'A} in
6    (* continue fold on None, exit fold when Some compare st. p(compare) *)
7    let predicate_step =
8      fun (ignore : Option 'A) => fun (x : 'A) =>
9      fun (recurse: Option 'A -> Option 'A) =>
10       let p_x = p x in
11       match p_x with
12       | True => Some {'A} x
13       | False => recurse init
14       end in
15   foldk predicate_step init
```

Like before, we take a type variable `'A` on line 2 and take the predicate on the next line. We begin by using this type variable to instantiate foldk, by giving it our processing type and return type. The processing type being the list element type and the result type being `Option 'A`. The next line is our accumulator, we assume that at the start of the search there is no satisfier.

On line 7, we write a fold description for foldk. This embodies the order of the recursion and conditions for recursion. `predicate_step` has the type `Option 'A -> 'A -> (Option 'A -> Option 'A) -> Option 'A`. The first argument is the accumulator, the second `x` is the next element to process and the third `recurse` is the recursive call. We do not care what the accumulator `ignore` is since if it mattered we will have already terminated.

On lines 10 to 12 check for `p x` and if so return `Some {'A} x`. In the case that `p x` does not hold, try again from scratch with the next element and so on via recursion. `recurse init` is in pseudo-code equal to λ`k. foldk predicate_step init k xs` where `xs` is the tail of our list of to be processed elements.

With the final line we partially apply `foldk` so that it just takes a list argument and gives us our final answer. The first argument of `foldk` gives us the specific fold we want, for example if you wanted a left fold you would replace `predicate_step` with something else.

3.4.5 Standard Libraries

The Scilla standard library contains five libraries: `BoolUtils.scilla`, `IntUtils.scilla`, `ListUtils.scilla`, `NatUtils.scilla` and `PairUtils.scilla`. As the names suggests these contracts implement utility operations for the `Bool`, `IntX`, `List`, `Nat` and `Pair` types, respectively.

To use functions from the standard library in a contract, the relevant library file must be imported using the `import` declaration. The following code snippet shows how to import the functions from the `ListUtils` and `IntUtils` libraries:

```
import ListUtils IntUtils
```

The `import` declaration must occur immediately before the contract's own library declaration, e.g.:

```
import ListUtils IntUtils

library WalletLib
... (* The declarations of the contract's own library values and functions *)

contract Wallet ( ... )
... (* The transitions and procedures of the contract *)
```

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

BoolUtils

- `andb : Bool -> Bool -> Bool`: Computes the logical AND of two `Bool` values.
- `orb : Bool -> Bool -> Bool`: Computes the logical OR of two `Bool` values.
- `negb : Bool -> Bool`: Computes the logical negation of a `Bool` value.
- `bool_to_string : Bool -> String`: Transforms a `Bool` value into a `String` value. `True` is transformed into `"True"`, and `False` is transformed into `"False"`.

IntUtils

- `intX_eq : IntX -> IntX -> Bool`: Equality operator specialised for each `IntX` type.

```
let int_list_eq = @list_eq Int64 in

let one = Int64 1 in
let two = Int64 2 in
let ten = Int64 10 in
let eleven = Int64 11 in

let nil = Nil {Int64} in
let l1 = Cons {Int64} eleven nil in
let l2 = Cons {Int64} ten l1 in
let l3 = Cons {Int64} two l2 in
let l4 = Cons {Int64} one l3 in

let f = int64_eq in
(* See if [2,10,11] = [1,2,10,11] *)
int_list_eq f l3 l4
```

- `uintX_eq : UintX -> UintX -> Bool`: Equality operator specialised for each `UintX` type.
- `intX_lt : IntX -> IntX -> Bool`: Less-than operator specialised for each `IntX` type.
- `uintX_lt : UintX -> UintX -> Bool`: Less-than operator specialised for each `UintX` type.
- `intX_neq : IntX -> IntX -> Bool`: Not-equal operator specialised for each `IntX` type.
- `uintX_neq : UintX -> UintX -> Bool`: Not-equal operator specialised for each `UintX` type.
- `intX_le : IntX -> IntX -> Bool`: Less-than-or-equal operator specialised for each `IntX` type.
- `uintX_le : UintX -> UintX -> Bool`: Less-than-or-equal operator specialised for each `UintX` type.
- `intX_gt : IntX -> IntX -> Bool`: Greater-than operator specialised for each `IntX` type.
- `uintX_gt : UintX -> UintX -> Bool`: Greater-than operator specialised for each `UintX` type.
- `intX_ge : IntX -> IntX -> Bool`: Greater-than-or-equal operator specialised for each `IntX` type.
- `uintX_ge : UintX -> UintX -> Bool`: Greater-than-or-equal operator specialised for each `UintX` type.

ListUtils

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

  Apply `f : 'A -> 'B` to every element of `l : List 'A`, constructing a list (of type `List 'B`) of the results.

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

(* Contract transition *)
(* Assume input is the list [ 1 ; 2 ; 3 ] *)
(* Apply f to all values in input *)
hash_list_int32 = @list_map Int32 ByStr32;
hashed_list = hash_list_int32 f input;
(* hashed_list is now [ sha256hash 1 ; sha256hash 2 ; sha256hash 3 ] *)
```

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

  Filter out elements on the list based on the predicate `f : 'A -> Bool`. If an element satisfies `f`, it will be in the resultant list, otherwise it is removed. The order of the elements is preserved.

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

(* Contract transition *)
(* Assume input is the list [ 1 ; 42 ; 2 ; 11 ; 12 ] *)
less_ten_int32 = @list_filter Int32;
less_ten_list = less_ten_int32 f l
(* less_ten_list is now  [ 1 ; 2 ]*)
```

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

  Return the head element of a list `l : List 'A` as an optional value. If `l` is not empty with the first element `h`, the result is `Some h`. If `l` is empty, then the result is `None`.

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

  Return the tail of a list `l : List 'A` as an optional value. If `l` is a non-empty list of the form `Cons h t`, then the result is `Some t`. If `l` is empty, then the result is `None`.

- `list_foldl_while : ('B -> 'A -> Option 'B) -> 'B -> List 'A -> 'B`

Given a function `f : 'B -> 'A -> Option 'B`, accumulator `z : 'B` and list `ls : List 'A` execute a left fold when our given function returns `Some x : Option 'B` using `f z x : 'B` or list is empty but in the case of `None : Option 'B` terminate early, returning `z`.

```
(* assume zero = 0, one = 1, negb is in scope and ls = [10,12,9,7]
 given a max and list with elements a_0, a_1, ..., a_m
 find largest n s.t. sum of i from 0 to (n-1) a_i <= max *)
let prefix_step = fun (len_limit : Pair Uint32 Uint32) => fun (x : Uint32) =>
  match len_limit with
  | Pair len limit => let limit_lt_x = builtin lt limit x in
    let x_leq_limit = negb limit_lt_x in
    match x_leq_limit with
    | True => let len_succ = builtin add len one in let l_sub_x = builtin sub limit x␣
↪in
      let res = Pair {Uint32 Uint32} len_succ l_sub_x in
      Some {(Pair Uint32 Uint32)} res
    | False => None {(Pair Uint32 Uint32)}
    end
  end in
let fold_while = @list_foldl_while Uint32 (Pair Uint32 Uint32) in
let max = Uint32 31 in
let init = Pair {Uint32 Uint32} zero max in
let prefix_length = fold_while prefix_step init ls in
match prefix_length with
| Pair length _ => length
end
```

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

  Append the first list to the front of the second list, keeping the order of the elements in both lists. Note that `list_append` has linear time complexity in the length of the first argument list.

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

  Return the reverse of the input list. Note that `list_reverse` has linear time complexity in the length of the argument list.

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

  Construct a list of all the elements in a list of lists. Each element (which has type `List 'A`) of the input list (which has type `List List 'A`) are all concatenated together, keeping the order of the input list. Note that `list_flatten` has linear time complexity in the total number of elements in all of the lists.

- `list_length : List 'A -> Uint32`

  Count the number of elements in a list. Note that `list_length` has linear time complexity in the number of elements in the list.

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

  Compare two lists element by element, using a predicate function `f : 'A -> 'A -> Bool`. If `f` returns `True` for every pair of elements, then `list_eq` returns `True`. If `f` returns `False` for at least one pair of elements, or if the lists have different lengths, then `list_eq` returns `False`.

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

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

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

(* Contract transition *)
(* Assume input is the list [ 1 ; 2 ; 3 ; 4 ] *)
keynumber = Int32 5;
list_mem_int32 = @list_mem Int32;
check_result = list_mem_int32 f keynumber input;
(* check_result is now False *)
```

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

  Check whether all elements of list `l : List 'A` satisfy the predicate `f : 'A -> Bool`. `list_forall` returns `True` if all elements satisfy `f`, and `False` if at least one element does not satisfy `f`.

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

  Check whether at least one element of list `l : List 'A` satisfies the predicate `f : 'A -> Bool`. `list_exists` returns `True` if at least one element satisfies `f`, and `False` if none of the elements satisfy `f`.

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

  Sort the input list `l : List 'A` using insertion sort. The comparison function `flt : 'A -> 'A -> Bool` provided must return `True` if its first argument is less than its second argument. `list_sort` has quadratic time complexity.

```
let int_sort = @list_sort Uint64 in

let flt =
  fun (a : Uint64) =>
```

(continues on next page)

(continued from previous page)

```
  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 = [ 3 ; 2 ; 1 ; 2 ; 3 ; 4 ; 2 ] *)
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

(* res1 = [ 1 ; 2 ; 2 ; 2 ; 3 ; 3 ; 4 ] *)
let res1 = int_sort flt l6
```

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

  Return the first element in a list `l : List 'A` satisfying the predicate `f : 'A -> Bool`. If at least one element in the list satisfies the predicate, and the first one of those elements is `x`, then the result is `Some x`. If no element satisfies the predicate, the result is `None`.

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

  Combine two lists element by element, resulting in a list of pairs. If the lists have different lengths, the trailing elements of the longest list are ignored.

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

  Combine two lists element by element using a combining function `f : 'A -> 'B -> 'C`. The result of `list_zip_with` is a list of the results of applying `f` to the elements of the two lists. If the lists have different lengths, the trailing elements of the longest list are ignored.

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

  Split a list of pairs into a pair of lists consisting of the elements of the pairs of the original list.

- `list_nth : Uint32 -> List 'A -> Option 'A.`

Return the element number `n` from a list. If the list has at least `n` elements, and the element number `n` is `x`, `list_nth` returns `Some x`. If the list has fewer than `n` elements, `list_nth` returns `None`.

NatUtils

- `nat_prev : Nat -> Option Nat`: Return the Peano number one less than the current one. If the current number is `Zero`, the result is `None`. If the current number is `Succ x`, then the result is `Some x`.
- `nat_fold_while : ('T -> Nat -> Option 'T) -> 'T -> Nat -> 'T`: Takes arguments `f : 'T -> Nat -> Option 'T`, `z : 'T` and `m : Nat`. This is `nat_fold` with early termination. Continues recursing so long as `f` returns `Some y` with new accumulator `y`. Once `f` returns `None`, the recursion terminates.
- `is_some_zero : Nat -> Bool`: Zero check for Peano numbers.
- `nat_eq : Nat -> Nat -> Bool`: Equality check specialised for the `Nat` type.
- `nat_to_int : Nat -> Uint32`: Convert a Peano number to its equivalent `Uint32` integer.
- `uintX_to_nat : UintX -> Nat`: Convert a `UintX` integer to its equivalent Peano number. The integer must be small enough to fit into a `Uint32`. If it is not, then an overflow error will occur.
- `intX_to_nat : IntX -> Nat`: Convert an `IntX` integer to its equivalent Peano number. The integer must be non-negative, and must be small enough to fit into a `Uint32`. If it is not, then an underflow or overflow error will occur.

PairUtils

- `fst : Pair 'A 'B -> 'A`: Extract the first element of a Pair.

```
let fst_strings = @fst String String in
let nick_name = "toby" in
let dog = "dog" in
let tobias = Pair {String String} nick_name dog in
fst_strings tobias
```

- `snd : Pair 'A 'B -> 'B`: Extract the second element of a Pair.

3.4.6 Scilla versions

Major and Minor versions

Scilla releases have a major version, minor version and a patch number, denoted as `X.Y.Z` where `X` is the major version, `Y` is the minor version, and `Z` the patch number.

- Patches are usually bug fixes that do not impact the behaviour of existing contracts. Patches are backward compatible.
- Minor versions typically include performance improvements and feature additions that do not affect the behaviour of existing contracts. Minor versions are backward compatible until the latest major version.
- Major versions are not backward compatible. It is expected that miners have access to implementations of each major version of Scilla for running contracts set to that major version.

Within a major version, miners are advised to use the latest minor revision.

The command `scilla-runner -version` will print major, minor and patch versions of the interpreter being invoked.

Contract Syntax

Every Scilla contract must begin with a major version declaration. The syntax is shown below:

```
(***************************************************)
(*                  Scilla version                 *)
(***************************************************)

scilla_version 0

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

library MyContractLib

...

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

contract MyContract

...
```

When deploying a contract the output of the interpreter contains the field `scilla_version : X.Y.Z`, to be used by the blockchain code to keep track of the version of the contract. Similarly, `scilla-checker` also reports the version of the contract on a successful check.

The `init.json` file

In addition to the version specified in the contract source code, it is also required that the contract's `init.json` specifies the same version when the contract is deployed and when the contract's transitions are invoked. This eases the process for the blockchain code to decide which interpreter to invoke.

A mismatch in the versions specified in `init.json` and the source code will lead to a gas-charged error by the interpreter.

An example `init.json`:

```
[
   {
      "vname" : "_creation_block",
      "type" : "BNum",
      "value" : "1"
   },
   {
      "vname" : "_scilla_version",
      "type" : "Uint32",
      "value" : "1",
   }
 ]
```

Chain Invocation Behaviour

Contracts of different Scilla versions may invoke transitions on each other.

The semantics of message passing between contracts is guaranteed to be backward compatible between major versions.

3.5 The Scilla checker

The Scilla checker (`scilla-checker`) works as a compiler frontend, parsing the contract and performing a number of static checks including typechecking.

3.5.1 Phases of the Scilla checker

The Scilla checker operates in distinct phases, each of which can perform checks (and potentially reject contracts that do not pass the checks) and add annotations to each piece of syntax:

- *Lexing and parsing* reads the contract code and builds an abstract syntax tree (AST). Each node in the tree is annotated with a location from the source file in the form of line and column numbers.
- *ADT checking* checks various constraints on user-defined ADTs.
- *Typechecking* checks that values in the contract are used in a way that is consistent with the type system. The typechecker also annotates each expression with its type.
- *Pattern-checking* checks that each pattern-match in the contract is exhaustive (so that execution will not fail due to no match being found), and that each pattern can be reached (so that the programmer does not inadvertently introduce a pattern branch that can never be reached).
- *Event-info* checks that messages and events in the contract contain all necessary fields. For events, if a contract emits two events with the same name (`_eventname`), then their structure (the fields and their types) must also be the same.
- *Cashflow analysis* analyzes the usage of variables and fields, and attempts to determine which fields are used to represent (native) blockchain money. No checks are performed, but expressions, variables and fields are annotated with tags indicating their usage.
- *Sanity-checking* performs a number of minor checks, e.g., that all parameters to a transition or a procedure have distinct names.

Annotations

Each phase in the Scilla checker can add an annotation to each node in the abstract syntax tree. The type of an annotation is specified through instantiations of the module signature `Rep`. `Rep` specifies the type `rep`, which is the type of the annotation:

```
module type Rep = sig
  type rep
  ...
end
```

In addition to the type of the annotation, the instantiation of `Rep` can declare helper functions that allow subsequent phases to access the annotations of previous phases. Some of these functions are declared in the `Rep` signature, because they involve creating new abstract syntax nodes, which must be created with annotations from the parser onwards:

```
module type Rep = sig
  ...

  val mk_id_address : string -> rep ident
  val mk_id_uint128 : string -> rep ident
  val mk_id_bnum    : string -> rep ident
  val mk_id_string  : string -> rep ident

  val rep_of_sexp : Sexp.t -> rep
  val sexp_of_rep : rep -> Sexp.t

  val parse_rep : string -> rep
  val get_rep_str: rep -> string
end
```

`mk_id_<type>` creates an identifier with an appropriate type annotation if annotation is one of the phases that has been executed. If the typechecker has not yet been executed, the functions simply create an (untyped) identifier with a dummy location.

`rep_of_sexp` and `sexp_of_rep` are used for pretty-printing. They are automatically generated if rep is defined with the `[@@deriving sexp]` directive.

`parse_rep` and `get_rep_str` are used for caching of typechecked libraries, so that they do not need to be checked again if they haven't changed. These will likely be removed in future versions of the Scilla checker.

As an example, consider the annotation module `TypecheckerERep`:

```
module TypecheckerERep (R : Rep) = struct
  type rep = PlainTypes.t inferred_type * R.rep
  [@@deriving sexp]

  let get_loc r = match r with | (_, rr) -> R.get_loc rr

  let mk_id s t =
    match s with
    | Ident (n, r) -> Ident (n, (PlainTypes.mk_qualified_type t, r))

  let mk_id_address s = mk_id (R.mk_id_address s) (bystrx_typ address_length)
  let mk_id_uint128 s = mk_id (R.mk_id_uint128 s) uint128_typ
  let mk_id_bnum    s = mk_id (R.mk_id_bnum s) bnum_typ
  let mk_id_string  s = mk_id (R.mk_id_string s) string_typ

  let mk_rep (r : R.rep) (t : PlainTypes.t inferred_type) = (t, r)

  let parse_rep s = (PlainTypes.mk_qualified_type uint128_typ, R.parse_rep s)
  let get_rep_str r = match r with | (_, rr) -> R.get_rep_str rr

  let get_type (r : rep) = fst r
end
```

The functor (parameterized structure) takes the annotation from the previous phase as the parameter `R`. In the Scilla checker this previous phase is the parser, but any phase could be added in-between the two phases by specifying the phase in the top-level runner.

The type `rep` specifies that the new annotation is a pair of a type and the previous annotation.

The function `get_loc` merely serves as a proxy for the `get_loc` function of the previous phase.

The function `mk_id` is a helper function for the `mk_id_<type>` functions, which create an identifier with the appropriate type annotation.

The `mk_rep` function is a helper function used by the typechecker.

Prettyprinting does not output the types of AST nodes, so the functions `parse_rep` and `get_rep_str` ignore the type annotations.

Finally, the function `get_type` provides access to type information for subsequent phases. This function is not mentioned in the `Rep` signature, since it is made available by the typechecker once type annotations have been added to the AST.

Abstract syntax

The `ScillaSyntax` functor defines the AST node types. Each phase will instantiate the functor twice, once for the input syntax and once for the output syntax. These two syntax instantiations differ only in the type of annotations of each syntax node. If the phase produces no additional annotations, the two instantiations will be identical.

The parameters `SR` and `ER`, both of typ `Rep`, define the annotations for statements and expressions, respectively.

```
module ScillaSyntax (SR : Rep) (ER : Rep) = struct

  type expr_annot = expr * ER.rep
  and expr = ...

  type stmt_annot = stmt * SR.rep
  and stmt = ...
end
```

Initial annotation

The parser generates the initial annotation, which only contains information about where the syntax node is located in the source file. The function `get_loc` allows subsequent phases to access the location.

The `ParserRep` structure is used for annotations both of statements and expressions.

```
module ParserRep = struct
  type rep = loc
  [@@deriving sexp]

  let get_loc l = l
  ...
end
```

Typical phase

Each phase that produces additional annotations will need to provide a new implementation of the `Rep` module type. The implementation should take the previous annotation type (as a structure implementing the `Rep` module type) as a parameter, so that the phase's annotations can be added to the annotations of the previous phases.

The typechecker adds a type to each expression node in the AST, but doesn't add anything to statement node annotations. Consequently, the typechecker only defines an annotation type for expressions.

In addition, the `Rep` implementation defines a function `get_type`, so that subsequent phases can access the type in the annotation.

```
module TypecheckerERep (R : Rep) = struct
  type rep = PlainTypes.t inferred_type * R.rep
  [@@deriving sexp]

  let get_loc r = match r with | (_, rr) -> R.get_loc rr

  ...
  let get_type (r : rep) = fst r
end
```

The Scilla typechecker takes the statement and expression annotations of the previous phase, and then instantiates `TypeCheckerERep` (creating the new annotation type), `ScillaSyntax` (creating the abstract syntax type for the previous phase, which serves as input to the typechecker), and `ScillaSyntax` again (creating the abstract syntax type that the typechecker outputs).

```
module ScillaTypechecker
  (SR : Rep)
  (ER : Rep) = struct

  (* No annotation added to statements *)
  module STR = SR
  (* TypecheckerERep is the new annotation for expressions *)
  module ETR = TypecheckerERep (ER)

  (* Instantiate ScillaSyntax with source and target annotations *)
  module UntypedSyntax = ScillaSyntax (SR) (ER)
  module TypedSyntax = ScillaSyntax (STR) (ETR)

  (* Expose target syntax and annotations for subsequent phases *)
  include TypedSyntax
  include ETR

  (* Instantiate helper functors *)
  module TU = TypeUtilities (SR) (ER)
  module TBuiltins = ScillaBuiltIns (SR) (ER)
  module TypeEnv = TU.MakeTEnv(PlainTypes)(ER)
  module CU = ScillaContractUtil (SR) (ER)
  ...
end
```

Crucially, the typechecker module exposes the annotations and the syntax type that it generates, so that they can be made available to the next phase.

The typechecker finally instantiates helper functors such as `TypeUtilities` and `ScillaBuiltIns`.

3.5.2 Cashflow Analysis

The cashflow analysis phase analyzes the usage of a contract's variables and fields, and attempts to determine which fields are used to represent (native) blockchain money. Each contract field is annotated with a tag indicating the field's usage.

The resulting tags are an approximation based on the usage of each contract field, and the usage of local variables in the contract. The tags are not guaranteed to be accurate, but are intended as a tool to help the contract developer use her fields in the intended manner.

Running the analysis

The cashflow analysis is activated by running `scilla-checker` with the option `-cf`. The analysis is not run by default, since it is only intended to be used during contract development.

A contract is never rejected due to the result of the cashflow analysis. It is up to the contract developer to determine whether the cashflow tags are consistent with the intended use of each contract field.

The Analysis in Detail

The analysis works by continually analysing the transitions and procedures of the contract until no further information is gathered.

The starting point for the analysis is the incoming message that invokes the contract's transition, the outgoing messages and events that may be sent by the contract, and any field being read from the blockchain such as the current blocknumber.

Both incoming and outgoing messages contain a field `_amount` whose value is the amount of money being transferred between accounts by the message. Whenever the value of the `_amount` field of the incoming message is loaded into a local variable, that local variable is tagged as representing money. Similarly, a local variable used to initialise the `_amount` field of an outgoing message is also tagged as representing money.

Conversely, the message fields `_sender`, `_recipient`, and `_tag`, the event field `_eventname`, and the blockchain field `BLOCKNUMBER` are known to not represent money, so any variable used to initialise those fields or to hold the value read from one of those fields is tagged as not representing money.

Once some variables have been tagged, their usage implies how other variables can be tagged. For instance, if two variables tagged as money are added to each other, the result is also deemed to represent money. Conversely, if two variables tagged as non-money are added, the result is deemed to represent non-money.

Tagging of contract fields happens when a local variable is used when loading or storing a contract field. In these cases, the field is deemed to have the same tag as the local variable.

Once a transition or procedure has been analyzed the local variables and their tags are saved, and the analysis proceeds to the next transition or procedure while keeping the tags of the contract fields. The analysis continues until all the transitions and procedures have been analysed without any existing tags having changed.

Tags

The analysis uses the following set of tags:

- *No information*: No information has been gathered about the variable. This sometimes (but not always) indicates that the variable is not being used, indicating a potential bug.
- *Money*: The variable represents money.
- *Not money*: The variable represents something other than money.
- *Map t* (where *t* is a tag): The variable represents a map or a function whose co-domain is tagged with *t*. Hence, when performing a lookup in the map, or when applying a function on the values stored in the map, the result is tagged with *t*. Keys of maps are assumed to always be *Not money*. Using a variable as a function parameter does not give rise to a tag.
- *Option t* (where *t* is a tag): The variable represents an option value, which, if it does not have the value `None`, contains the value `Some x` where `x` has tag *t*.
- *Pair t1 t2* (where *t1* and *t2* are tags): The variable represents a pair of values with tags *t1* and *t2*, respectively.
- *Inconsistent*: The variable has been used to represent both money and not money. Inconsistent usage indicates a bug.

Lists and user-defined ADTs are currently not supported.

Library and local functions are only partially supported, since no attempt is made to connect the tags of parameters to the tag of the result. Built-in functions are fully supported, however.

Example

As an example, consider a crowdfunding contract written in Scilla. Such a contract may declare the following immutable parameters and mutable fields:

```
contract Crowdfunding

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

(* Mutable fields *)
field backers : Map ByStr20 Uint128 = ...
field funded : Bool = ...
```

The `owner` parameter represents the address of the person deploying the contract. The `goal` parameter is the amount of money the owner is trying to raise, and the `max_block` parameter represents the deadline by which the goal is to be met.

The field `backers` is a map from the addresses of contributors to the amount of money contributed, and the field `funded` represents whether the goal has been reached.

Since the field `goal` represents an amount of money, `goal` should be tagged as *Money* by the analysis. Similarly, the `backers` field is a map with a co-domain representing *Money*, so `backers` should be tagged with *Map Money*.

Conversely, both `owner`, `max_block` and `funded` represent something other than money, so they should all be tagged with *Not money*.

The cashflow analysis will tag the parameters and fields according to how they are used in the contract's transitions and procedures, and if the resulting tags do not correspond to the expectation, then the contract likely contains a bug somewhere.

3.6 Interpreter Interface

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

3.6.1 Calling Interface

A transition defined in a contract can be called either by the issuance of a transaction, or by message calls from another 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 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

3.6.2 Initializing the Immutable State

`init.json` defines the values of the immutable parameters 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",
        "type" : "ByStr20",
        "value" : "0x1234567890123456789012345678901234567890"
    },
    {
        "vname" : "_this_address",
        "type" : "ByStr20",
        "value" : "0xabfeccdc90123456789012345678 90f777567890"
```

(continues on next page)

(continued from previous page)

```
    },
    {
        "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" : "_this_address",
      "type" : "ByStr20",
      "value" : "0xabfeccdc9012345678901234567890f777567890"
  },
  {
      "vname" : "goal",
      "type" : "Uint128",
      "value" : "500000000000000"
  },
  {
      "vname" : "_creation_block",
      "type" : "BNum",
      "value" : "1"
  }
]
```

3.6.3 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: At the moment, the only blockchain value that is exposed to contracts is the current `BLOCKNUMBER`.

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

3.6.4 Input Message

`input_message.json` contains the information required 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" : "0x643456789012345678901234567890123456 78cd",
  "params"  : [
    {
      "vname" : "from",
      "type"  : "ByStr20",
      "value" : "0x1234567890123456789012345678901234567890"
    },
    {
      "vname" : "to",
      "type"  : "ByStr20",
      "value" : "0x7834567890123456789012345678901234567 8cd"
    },
    {
      "vname" : "tokens",
      "type"  : "Uint128",
      "value" : "500000000000000"
    }
  ]
}
```

3.6.5 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"` or `"false"`)
message	The message to be sent to another contract/non-contract account, if any.
states	An array of objects that form the new contract state
events	An array of events emitted by the transition and the procedures it invoked.

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

Field	Description
_tag	Transition to be invoked
_amount	Number of QAs (10^-12 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`).

- `events` is an array of objects that represents the events emitted by the transition. The fields in each object in the `events` array are given below:

Field	Description
_eventname	The name of the event
params	An array of additional event fields

The `params` array is encoded similar to how `init.json` is encoded, with each parameter specifying the (`vname`, `type`, `value`) of each event field.

Example 1

An example of the output generated by `Crowdfunding.scilla` is given below. The example also shows the format for maps in contract states.

```
{
  "scilla_major_version": "0",
  "gas_remaining": "7365",
  "_accepted": "false",
  "message": {
    "_tag": "",
    "_amount": "100000000000000",
    "_recipient": "0x1234567890123456789012345678901234567890ab",
    "params": []
  },
  "states": [
    { "vname": "_balance", "type": "Uint128", "value": "300000000000000" },
    {
      "vname": "backers",
      "type": "Map (ByStr20) (Uint128)",
      "value": [
        { "key": "0x1234567890123456789012345678901234567890cd", "val": "200000000000000
↪" },
        { "key": "0x1234567890123456789012345678901234567890abcd", "val": "100000000000000
↪" }
      ]
    },
    {
      "vname": "funded",
      "type": "Bool",
      "value": { "constructor": "False", "argtypes": [], "arguments": [] }
    }
  ],
  "events": [
    {
      "_eventname": "ClaimBackSuccess",
      "params": [
        {
          "vname": "caller",
          "type": "ByStr20",
          "value": "0x1234567890123456789012345678901234567890ab"
        },
        { "vname": "amount", "type": "Uint128", "value": "100000000000000" },
        { "vname": "code", "type": "Int32", "value": "9" }
      ]
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

Example 2

For values of an ADT type, the `value` field contains three subfields:

- `constructor`: The name of the constructor used to construct the value.
- `argtypes`: An array of type instantiations. For the `List` and `Option` types, this array will contain one type, indicating the type of the list elements or the optional value, respectively. For the `Pair` type, the array will contain two types, indicating the types of the two values in the pair. For all other ADTs, the array will be empty.
- `arguments`: The arguments to the constructor.

The following example shows how values of the `List` and `Option` types are represented in the output json:

```
{
  "scilla_major_version": "0",
  "gas_remaining": "7733",
  "_accepted": "false",
  "message": null,
  "states": [
    { "vname": "_balance", "type": "Uint128", "value": "0" },
    {
      "vname": "gpair",
      "type": "Pair (List (Int64)) (Option (Bool))",
      "value": {
        "constructor": "Pair",
        "argtypes": [ "List (Int64)", "Option (Bool)" ],
        "arguments": [
          [],
          { "constructor": "None", "argtypes": [ "Bool" ], "arguments": [] }
        ]
      }
    },
    { "vname": "llist", "type": "List (List (Int64))", "value": [] },
    { "vname": "plist", "type": "List (Option (Int32))", "value": [] },
    {
      "vname": "gnat",
      "type": "Nat",
      "value": { "constructor": "Zero", "argtypes": [], "arguments": [] }
    },
    {
      "vname": "gmap",
      "type": "Map (ByStr20) (Pair (Int32) (Int32))",
      "value": [
        {
          "key": "0x1234567890123456789012345678901234567890ab",
          "val": {
            "constructor": "Pair",
            "argtypes": [ "Int32", "Int32" ],
            "arguments": [ "1", "2" ]
          }
        }
```

(continues on next page)

(continued from previous page)

```
      ]
    }
  ],
  "events": []
}
```

3.6.6 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": "0x1234567890123456789012345678901234567890",
        "val": "200000000000000"
      },
      {
        "key": "0x1234567890123456789012345678901234567891",
        "val": "100000000000000"
      }
    ]
  },
  {
    "vname": "funded",
    "type": "Bool",
    "value": {
      "constructor": "False",
      "argtypes": [],
      "arguments": []
    }
  },
  {
    "vname": "_balance",
    "type": "Uint128",
    "value": "300000000000000"
  }
]
```

3.7 Contact

Questions? Ask us on our Gitter Channel.